6.1800 Spring 2025

Lecture #22: Low-level Exploits

smashing stacks, trusting trust

Katrina LaCurts | lacurts@mit.edu | 6.1800 2025

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q
[illegible]	[illegible]	[illegible]

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

last time, our threat model allowed for an adversary that had access to some sensitive data stored on our machine

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

last time, our threat model allowed for an adversary that had access to some sensitive data stored on our machine

a straightforward adversary in this case is someone like a system administrator, who is intended to have access to this data

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

last time, our threat model allowed for an adversary that had access to some sensitive data stored on our machine

a straightforward adversary in this case is someone like a system administrator, who is intended to have access to this data

today, we'll look at how an adversary that is *not* intended to have access to this data might get it

username	salt	slow_hash(password|salt)
user1	TU6kbcuPm7jA./IQYZG.8O	rBda9fbnXhUCWi6c9MjlUtQFlK1I4Sq
user2	y7oSC2QsrvXTzEZlDZFdwu	tjFcpSrZN6ryOYueyrAtUfFnFaOui2C
user3	4ncYRSB5v3rWiU1nPpA0iu	hacrgRlfU44c9XnBckef2fu.ifuB.Ya
user4	SK9H4x4Ha0Owz4NOTwj20.	wWk3GjGeMspoqy3VcMghpbkE5OjHQXS
user5	j8YyeDX.9GnsT5Hu94z7tO	ViflhwGHl.5H3j0mawzBPdKTiXf5L6.
user6	CIqY72CGM8KNQId1CqXY7.	stk3mDJDaaH9Nfgf/ePJrkRoK15.Heu
user7	OGtMXrEZEx0L544Odvrhbe	A.7NaJc21Y6I3J6rdJtiIJXVpMvaMgG
user8	RFeT9TVol8cmpQdhqMCV5.	yVzcp0OjXBoNjcMWHpAxVulFqdM5W9m
user9	rDAhDK5V6n3TUS3ahf2Z9e	Af4wBH1YqLTvxrhBgVGP85IALXRya3C
user10	nvOYvT0/oczOW5lmbVZSUO	b4miFmYcRy0/TFVhttntbrrLPLjFDKu
user11	yNL/e3PpBsfBYgwi0Ai/gu	bbT5sTcmsklsyXVILfVdJ/HAIEonb..
user12	1zroUl0scwDzgG3GY86pFO	MG5LtQ6m/c4gVxbLalpPIJ4O3eXFPry
user13	TAkv7nBQ5amY4V.aIjez0u	LHPo8.0XJDGleWWgG87nPvY8/vNPa2G
user14	1J796dTzufUC8ItVIKIyOu	pAI7ZRWvVOhxBVW/sttFquJCl/74LTC
user15	/x.Vk/XhUILbk3XjgyVyfO	zx1P3YgW8d9m1n9lZ6GW7jsbBALniWi
user16	hyg8T0JPDX3dCf92Zkx4Yu	5Oh.8uSUrokBgqnByYYH/mDEH7my98C
user17	YbaYOSdkA01IF.drWa6CXO	ZKbZQtEh4UNoTflWsXs9hZ7wbnnzgC.
user18	yaE.gULeQg.K2SelX191Q.	E/syZIC.1.zg5.ZTMZwWX/RmkvpipNu
user19	NLt0SA/QPo2IIbtb7G56lO	eOX2p48XcKRXKFY87f56h3W.UEeO7Gi
user20	RFFSWUGGFeX5XNyW8rLToe	0W94ciFDN5stvqVzYsli4t/SNA2pwhS
user21	YWEgwinWuKrNUFvgzQKUNe	yatU0vWN//72Ul8OdxGHnClTLWdTfXe
user22	ukqUgoOZWCqIQjH3DwC4xe	jg1.OSatbZooR6l4taWv3HBpXNN5Xp2
user23	sPRFpmFnu5G4lAPUkV0wrO	mVpzAYGXEgs583nG894R98k1S3YmP1q

policy: provide **authentication** for users

threat model: adversary has access to the entire stored table

last time, our threat model allowed for an adversary that had access to some sensitive data stored on our machine

a straightforward adversary in this case is someone like a system administrator, who is intended to have access to this data

today, we'll look at how an adversary that is *not* intended to have access to this data might get it

our threat model for most of today is an adversary with the ability to run code on our machine, but not necessarily any particular privileges (e.g., root access)

```
void function(int a) {
  int y = a + 2;
  // do whatever
}

void main() {
  int x = 0;
  function(7);
  x = 5;
  // maybe other stuff here
}
```

```
void function(int a) {
  int y = a + 2;
  // do whatever
}

void main() {
  int x = 0;
  function(7);
  x = 5;
  // maybe other stuff here
}
```

the program stack needs to enable a few things:

```
void function(int a) {
  int y = a + 2;
  // do whatever
}

void main() {
  int x = 0;
  function(7);
  x = 5;
  // maybe other stuff here
}
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute

```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute

```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer (“frame pointer”)
```

```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute

```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer ("frame pointer")
```

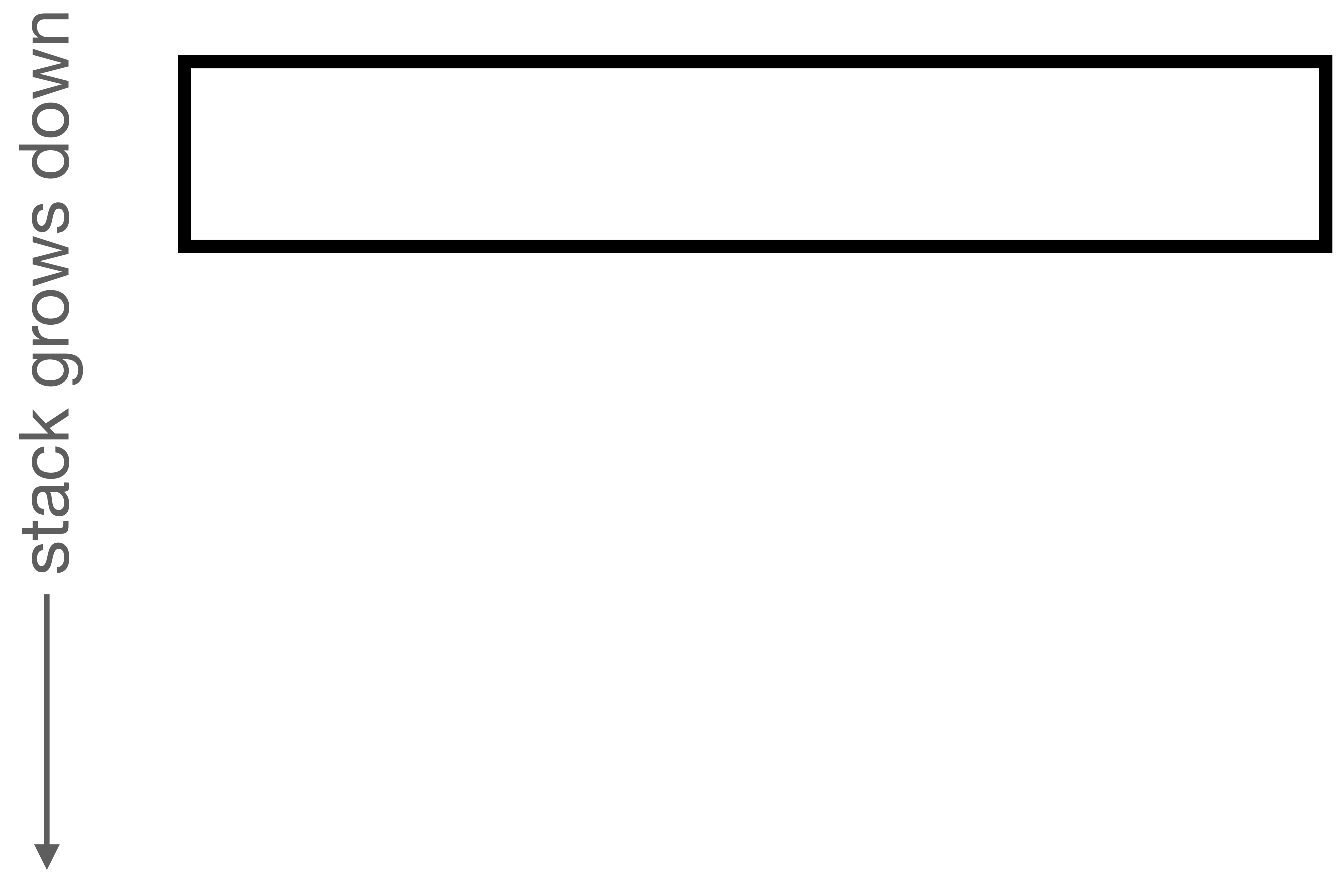
```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



`IP` = Instruction pointer
`SP` = Stack pointer
`BP` = Base pointer (“frame pointer”)

```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute

stack grows down

local vars in **`main`**

`IP` = Instruction pointer

`SP` = Stack pointer

`BP` = Base pointer ("frame pointer")

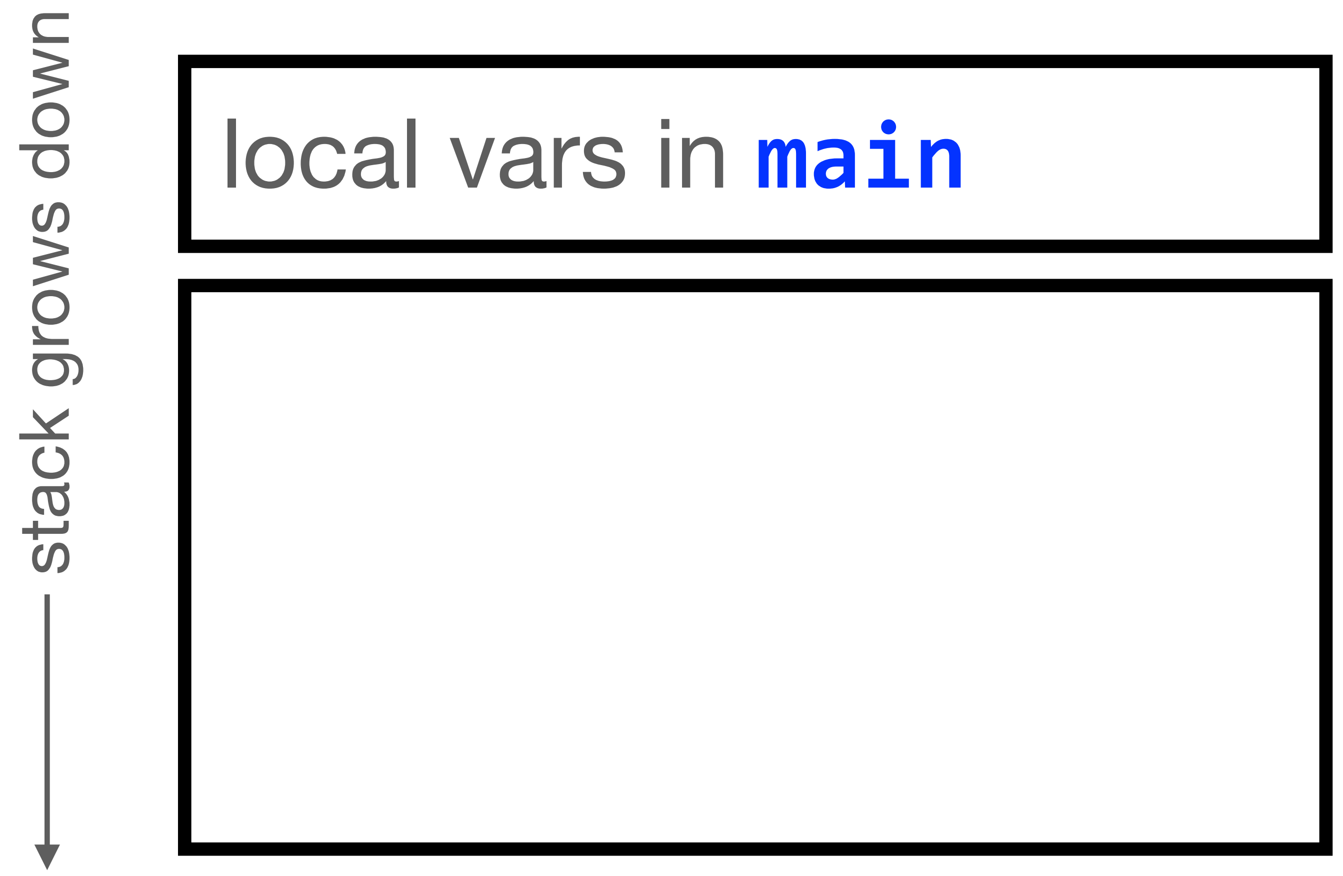
```c
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



`IP` = Instruction pointer
`SP` = Stack pointer
`BP` = Base pointer (“frame pointer”)

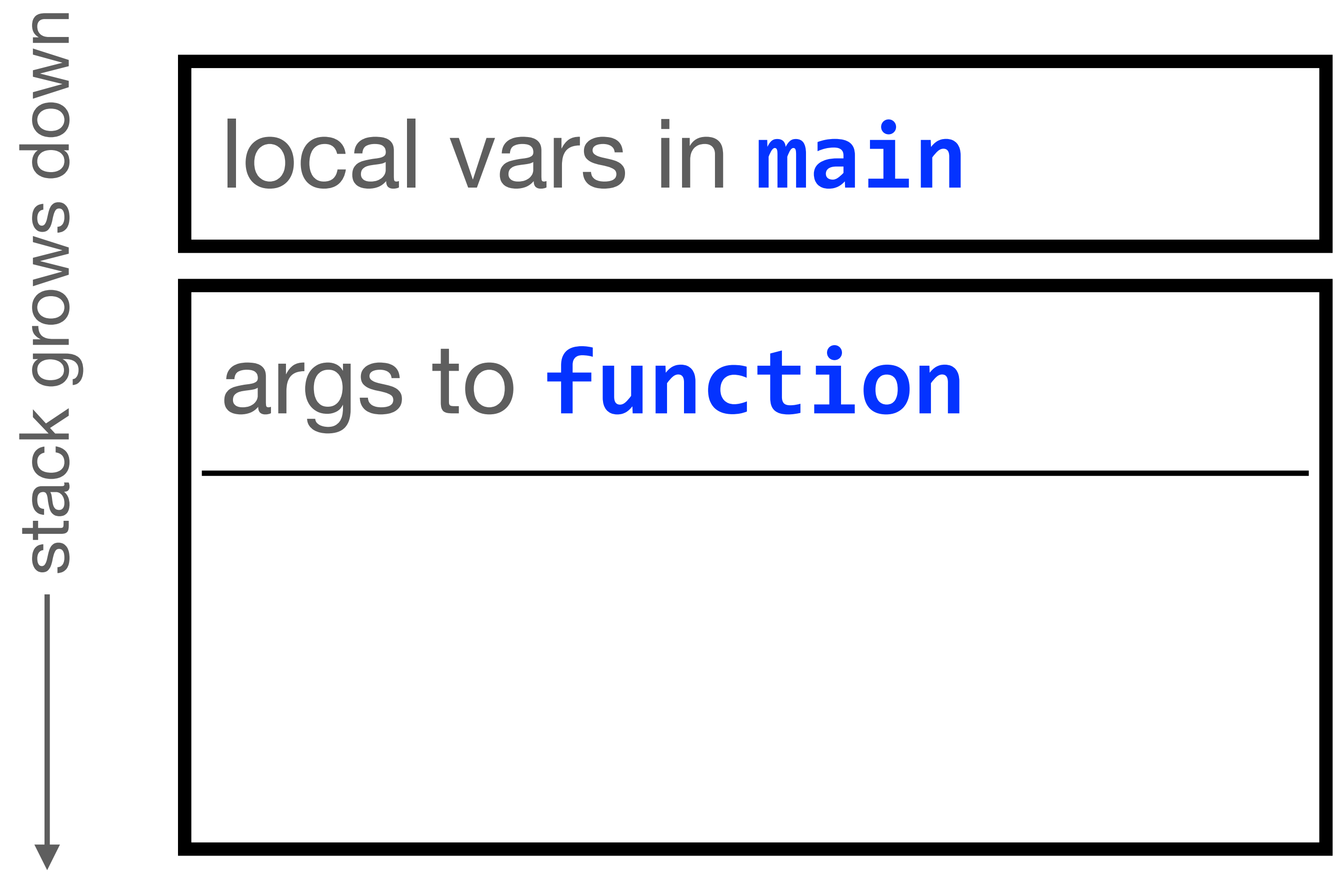
```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



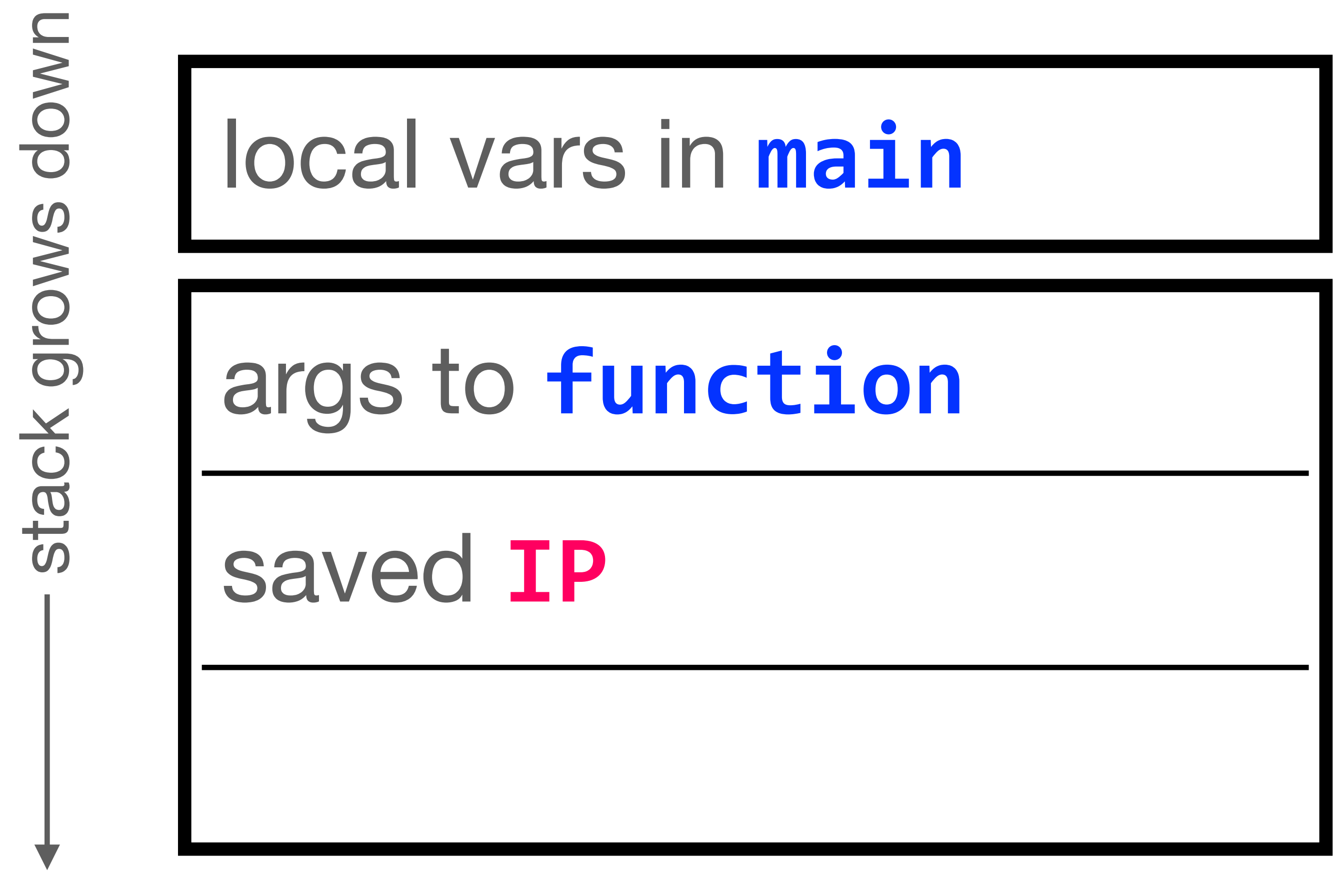
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer (“frame pointer”)

```c
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.
2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



`IP = Instruction pointer`
`SP = Stack pointer`
`BP = Base pointer (“frame pointer”)`

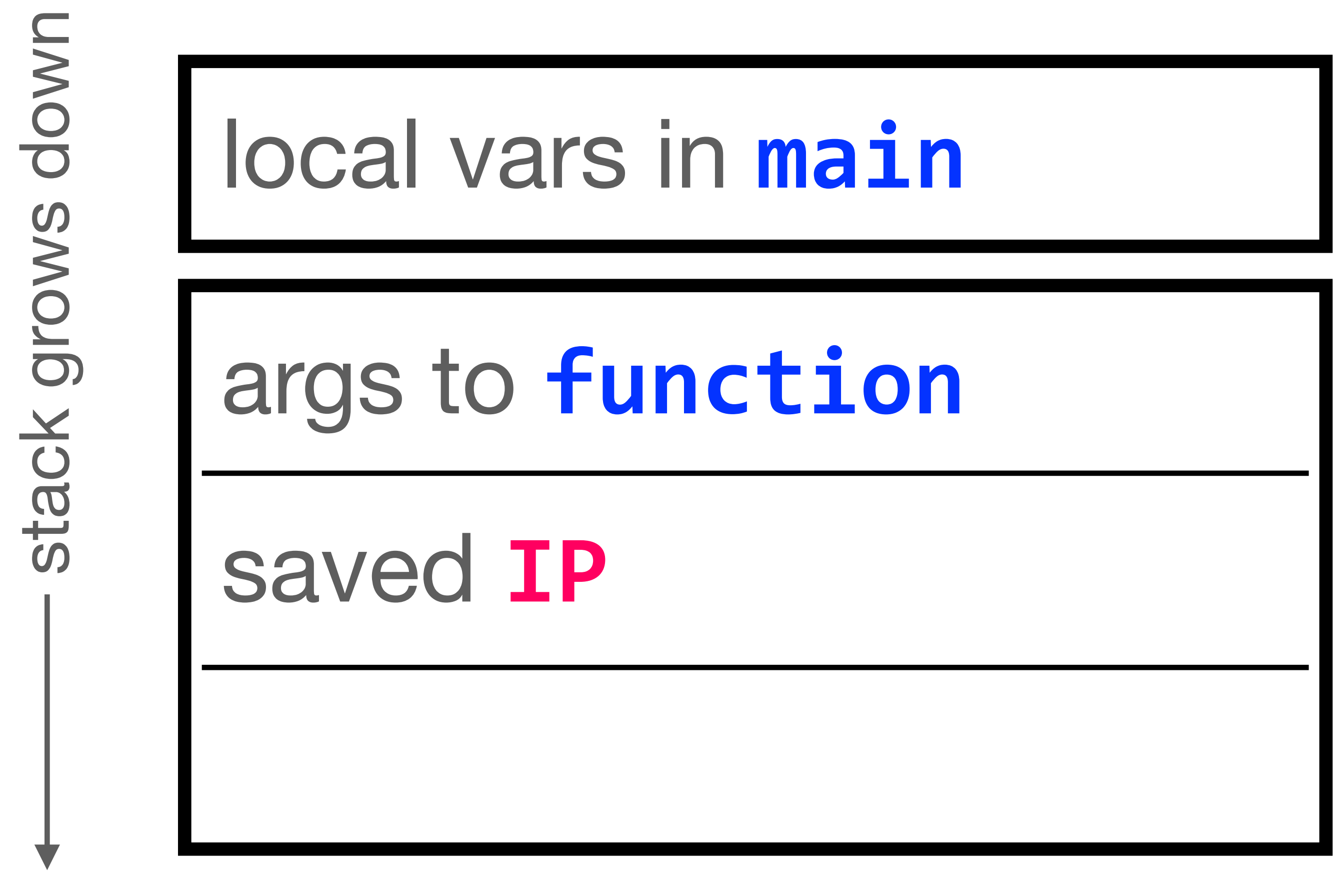
```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



the saved IP will let the code return to line 9 of `main()` after `function()` ends

```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer ("frame pointer")
```

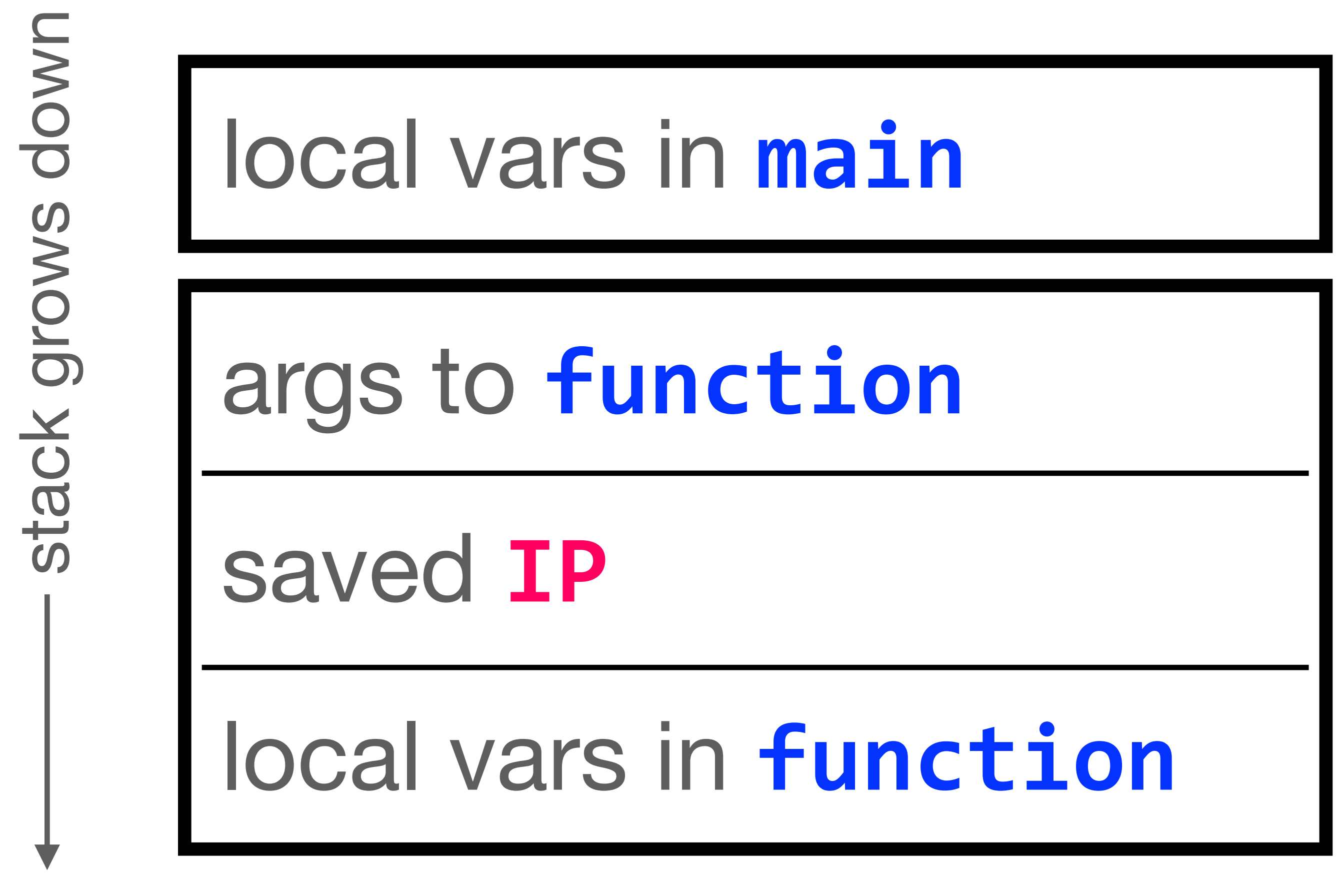
```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



the saved IP will let the code return to line 9 of `main()` after `function()` ends

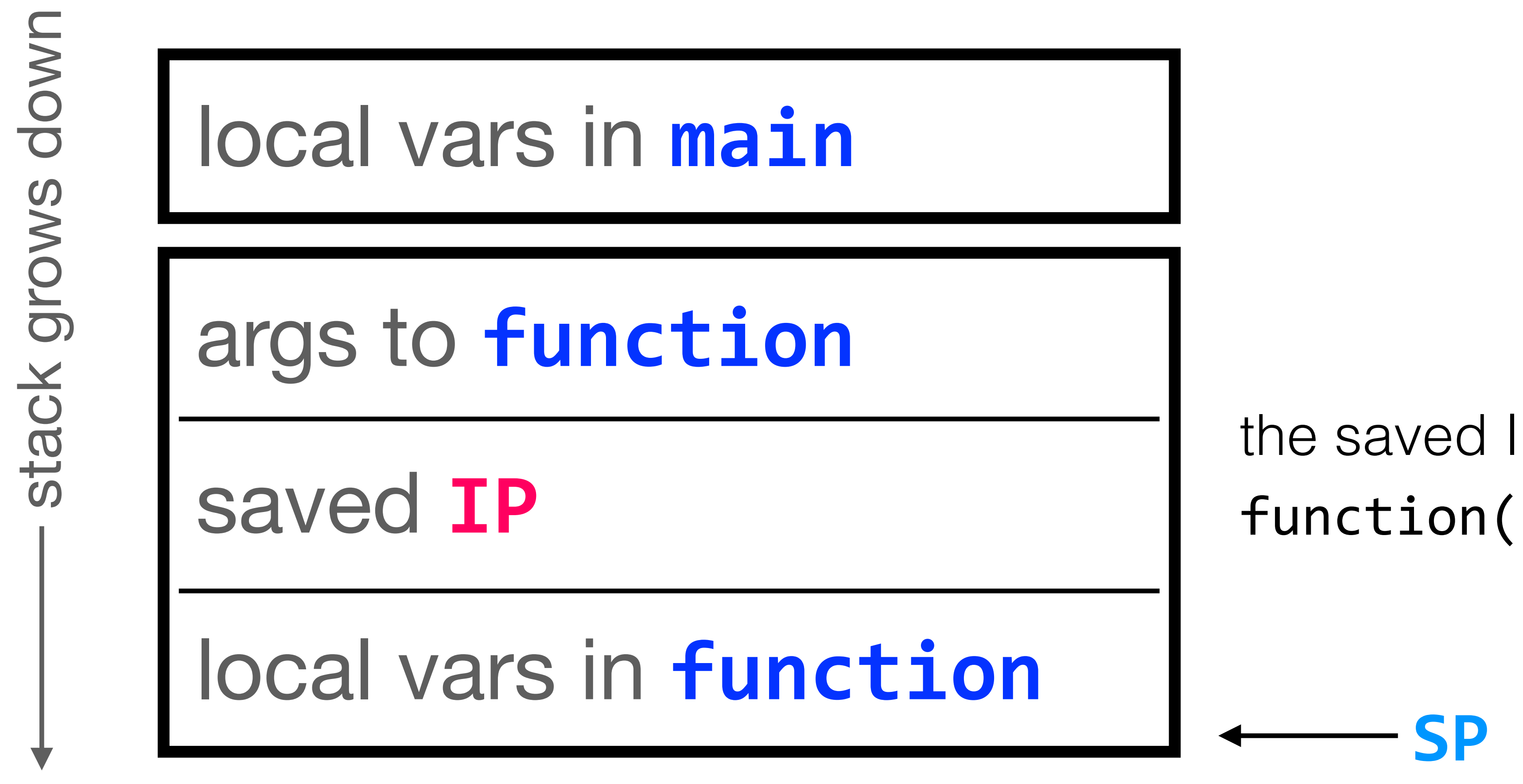
```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer (“frame pointer”)
```

```c
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.
2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



the saved IP will let the code return to line 9 of `main()` after `function()` ends

```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer (“frame pointer”)
```

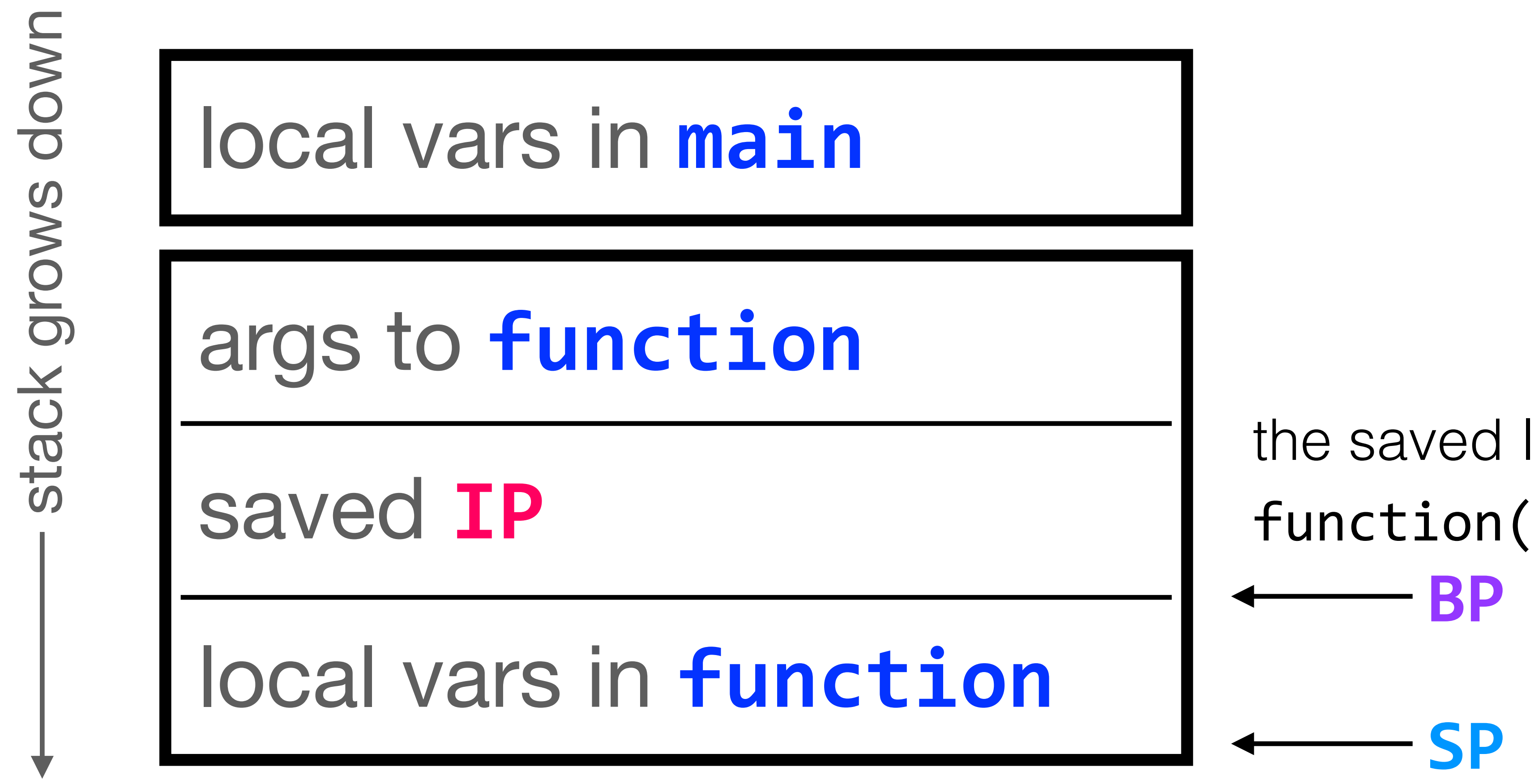
```c
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



the saved IP will let the code return to line 9 of `main()` after `function()` ends

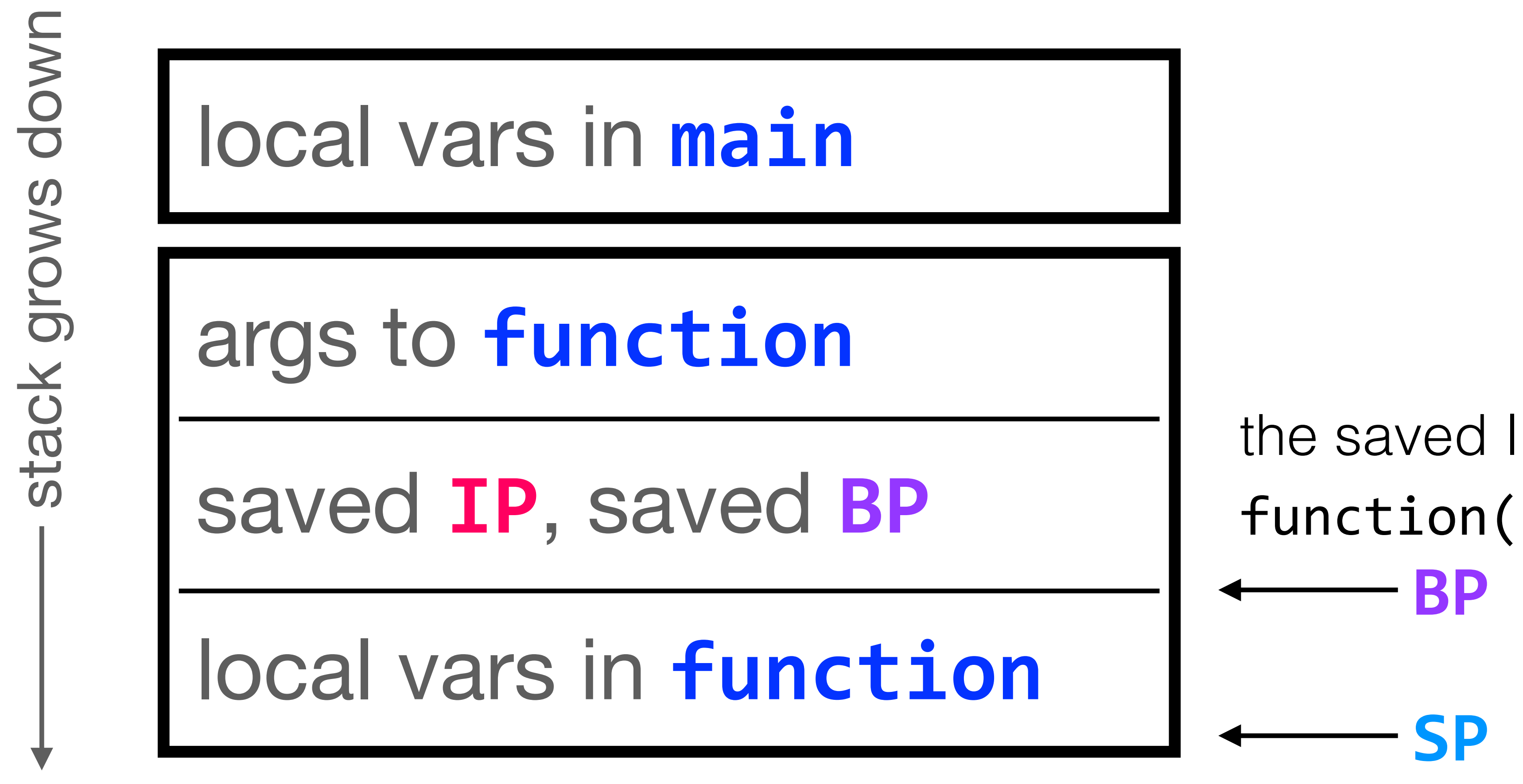
```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer (“frame pointer”)
```

```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

1. each function should be able to access its own local variables, including any arguments passed to it.
2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute



the saved IP will let the code return to line 9 of `main()` after `function()` ends

```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer (“frame pointer”)
```

```
1:  void function(int a) {
2:    int y = a + 2;
3:    // do whatever
4:  }
5:
6:  void main() {
7:    int x = 0;
8:    function(7);
9:    x = 5;
10:   // maybe other stuff here
11: }
```

the program stack needs to enable a few things:

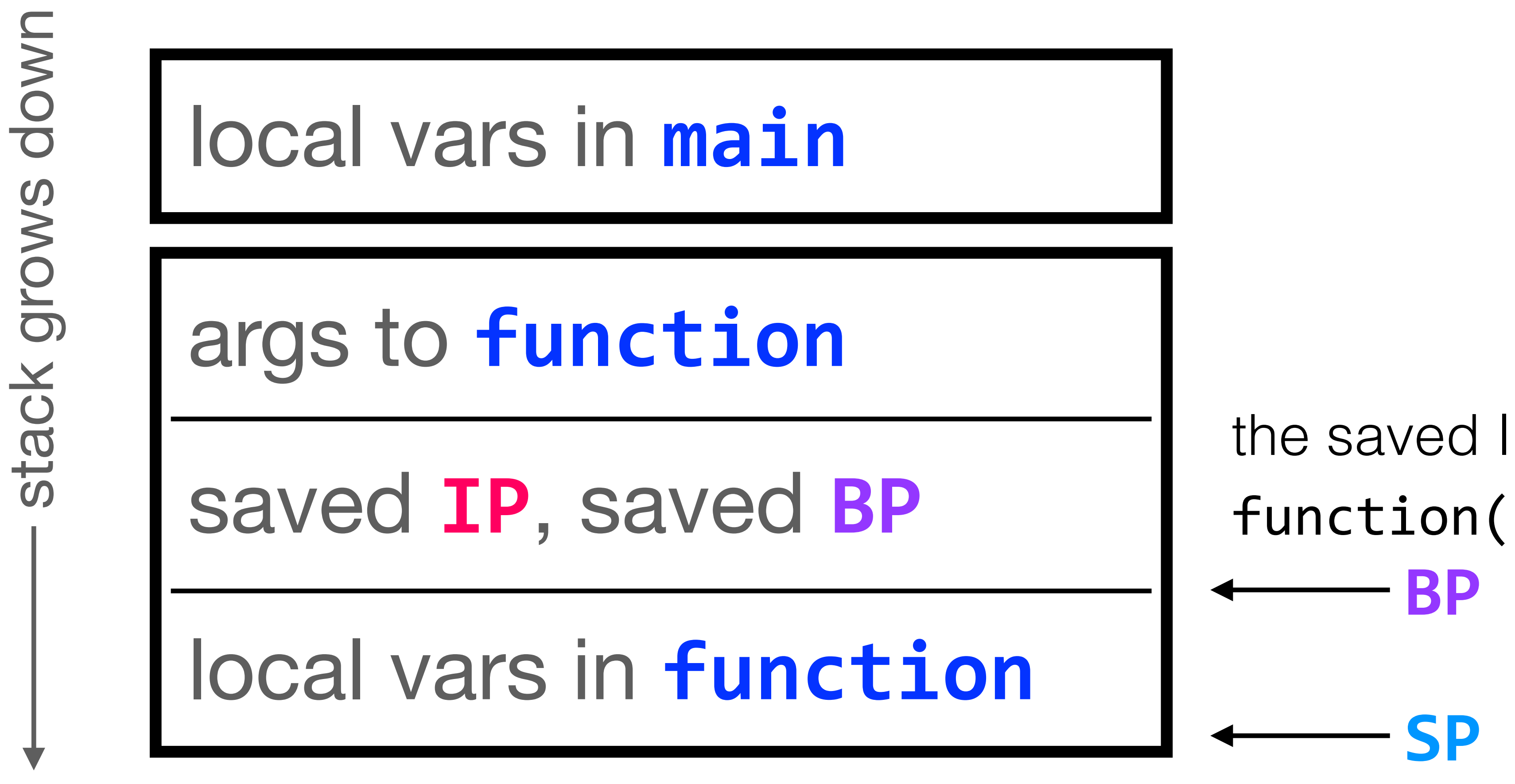
1. each function should be able to access its own local variables, including any arguments passed to it.

2. after a function returns, the next line of the calling function should execute

   here that means that once the call to `function()` returns, the next line in `main()` — line 9 — should execute

to return to **`main()`** after **`function()`** ends, we use **`BP`** to locate the start of the current stack frame. the previous values of **`BP`** and **`IP`** are located at a fixed offset from that, so we can reset **`BP`** and **`IP`**, and continue on.

`IP` will now point to the next instruction in **`main()`**, and **`BP`** will point to the start of **`main()`**'s stack frame.



the saved IP will let the code return to line 9 of `main()` after `function()` ends

```
IP = Instruction pointer
SP = Stack pointer
BP = Base pointer (“frame pointer”)
```

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer); // sort of like input() in python

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```

args to **main**
saved **IP**, saved **BP**
modified (4 bytes) **buffer** (64 bytes)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  volatile int modified;
  char buffer[64];

  modified = 0;
  gets(buffer); // sort of like input() in python

  if(modified != 0) {
      printf("you have changed the 'modified' variable\n");
  } else {
      printf("Try again?\n");
  }
}
```

adversary's goal: input a string that overwrites `modified`

args to `main`
saved `IP`, saved `BP`
`modified` (4 bytes) `buffer` (64 bytes)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
      printf("calling function pointer, jumping to 0x%08x\n", fp);
      fp();
  }
}
```

args to **function**
saved **IP**, saved **BP**
fp (4 bytes) **buffer** (64 bytes)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  volatile int (*fp)();
  char buffer[64];

  fp = 0;

  gets(buffer);

  if(fp) {
      printf("calling function pointer, jumping to 0x%08x\n", fp);
      fp();
  }
}
```

adversary's goal: input a string that overwrites `fp` so that the code jumps into `win`

args to `function`
saved `IP`, saved `BP`
`fp` (4 bytes) `buffer` (64 bytes)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];
  gets(buffer);
}
```

args to **function**
saved **IP**, saved **BP**
buffer (64 bytes)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];
  gets(buffer);
}
```

adversary's goal: input a string that overwrites the saved `IP` so that the code jumps into `win`

args to `function`
saved `IP`, saved `BP`
`buffer` (64 bytes)

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];
  gets(buffer);
}
```

adversary's goal: input a string that overwrites the saved **IP** so that the code jumps into `win`

args to `function`
saved `IP`, saved `BP`
`buffer` (64 bytes)

BP →

SP →

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
  printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
  char buffer[64];
  gets(buffer);
}
```

adversary's goal: input a string that overwrites the saved **IP** so that the code jumps into `win`

args to `function`
saved **IP**, saved **BP**
`buffer` (64 bytes)

BP →
SP →

in the demo, there is a bit of extra space between buffer and the saved IP

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

example counter-attacks: arc-injection ("return-to-libc"), heap smashing, pointer subterfuge

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

example counter-attacks: arc-injection (“return-to-libc”), heap smashing, pointer subterfuge

question: you can’t perform stack-smashing attacks with a language like Python. **why not?**

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

example counter-attacks: arc-injection ("return-to-libc"), heap smashing, pointer subterfuge

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

example counter-attacks: arc-injection (“return-to-libc”), heap smashing, pointer subterfuge

bounds-checking is one solution, but it ruins the ability to create compact C code (note the trade-off of **security vs. performance**)

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

example counter-attacks: arc-injection (“return-to-libc”), heap smashing, pointer subterfuge

bounds-checking is one solution, but it ruins the ability to create compact C code (note the trade-off of **security vs. performance**)

```
struct record {
  int age;
  int sal;
  char name[1];
};

struct record *r;
char buf[100];
read(socket, buf, 100)
r = (struct record *)buf;
printf ("%d,%d,%s\n",r->age,r->sal,r->name);
```

modern linux has **protections** in place to prevent the attacks on the previous slides, but there are **counter-attacks** to those protections

example protections: non-executable stacks, address space layout randomization, etc.

example counter-attacks: arc-injection (“return-to-libc”), heap smashing, pointer subterfuge

bounds-checking is one solution, but it ruins the ability to create compact C code (note the trade-off of **security vs. performance**)

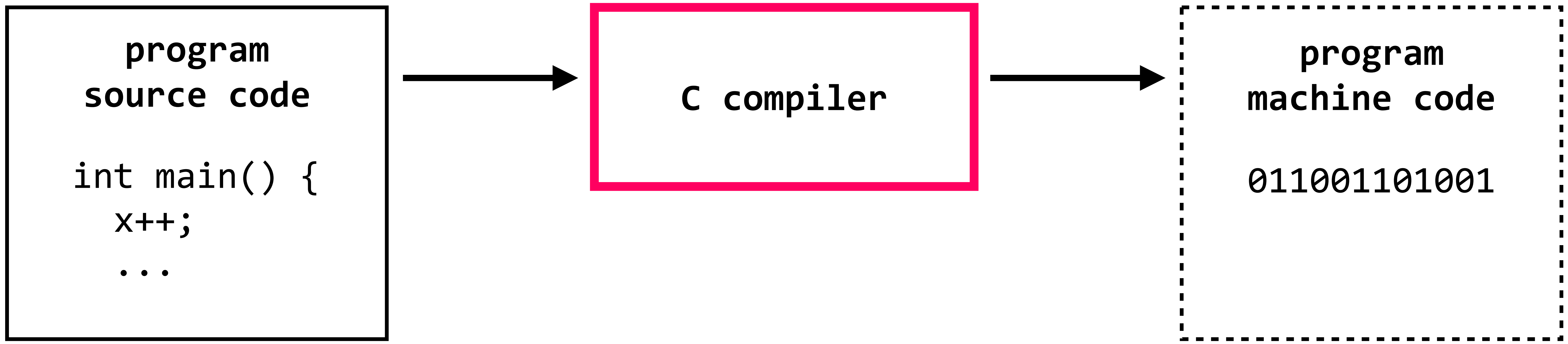
```
struct record {
  int age;
  int sal;
  char name[1];
};

struct record *r;
char buf[100];
read(socket, buf, 100)
r = (struct record *)buf;
printf ("%d,%d,%s\n",r->age,r->sal,r->name);
```

for example, here is some network I/O code in C (exactly what it does doesn’t matter at all for this example). this generates very compact assembly, and takes hundreds of lines in Java.

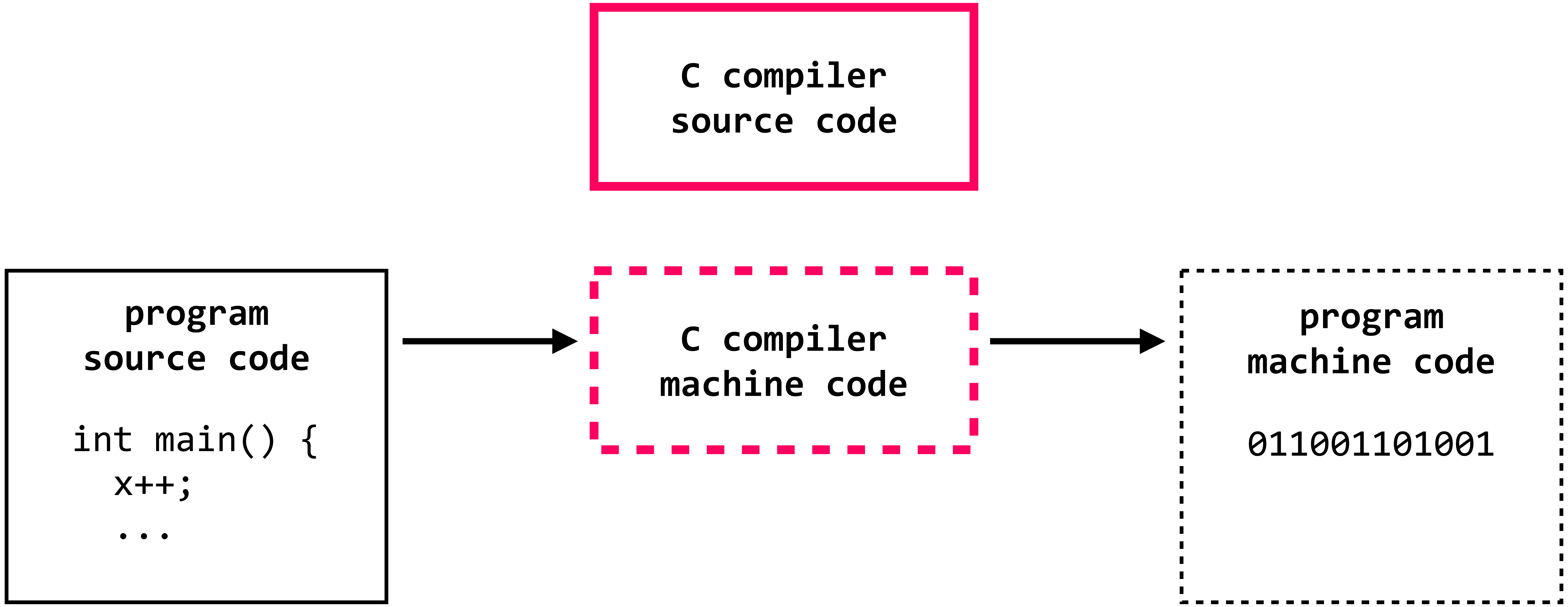
compilers: can we trust them?

compilers take source code as an input, and output machine code



compilers: can we trust them?

compilers take source code as an input, and output machine code



compilers: can we trust them?

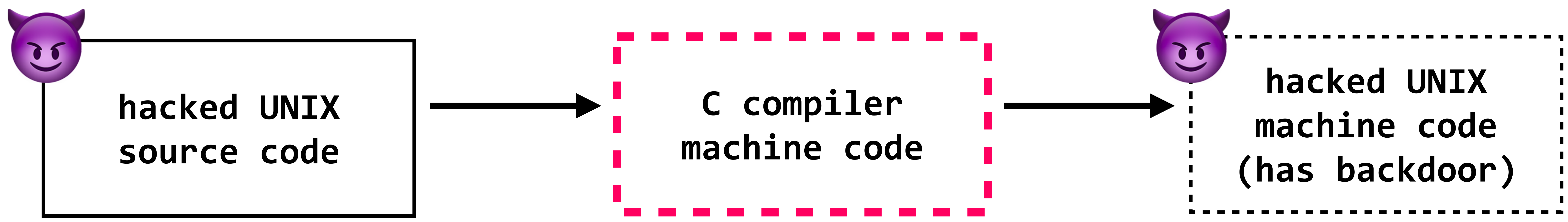
compilers take source code as an input, and output machine code

compilers: can we trust them?

compilers take source code as an input, and output machine code

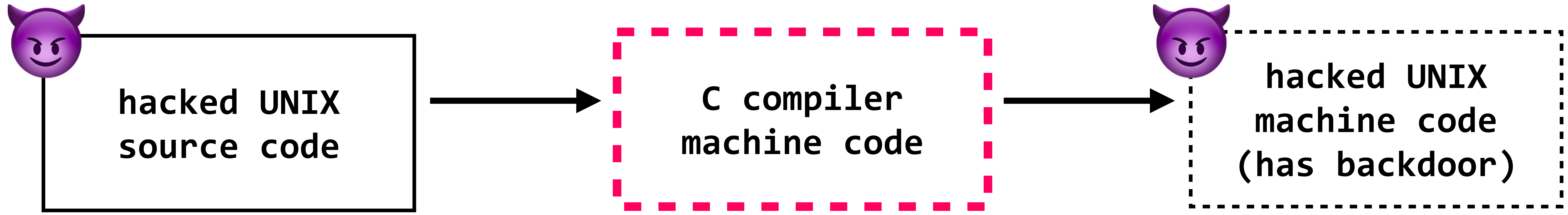
compilers: can we trust them?

compilers take source code as an input, and output machine code



compilers: can we trust them?

compilers take source code as an input, and output machine code



this backdoor is easily discovered in the hacked UNIX source

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked C compiler has code that
inserts a backdoor into UNIX

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code

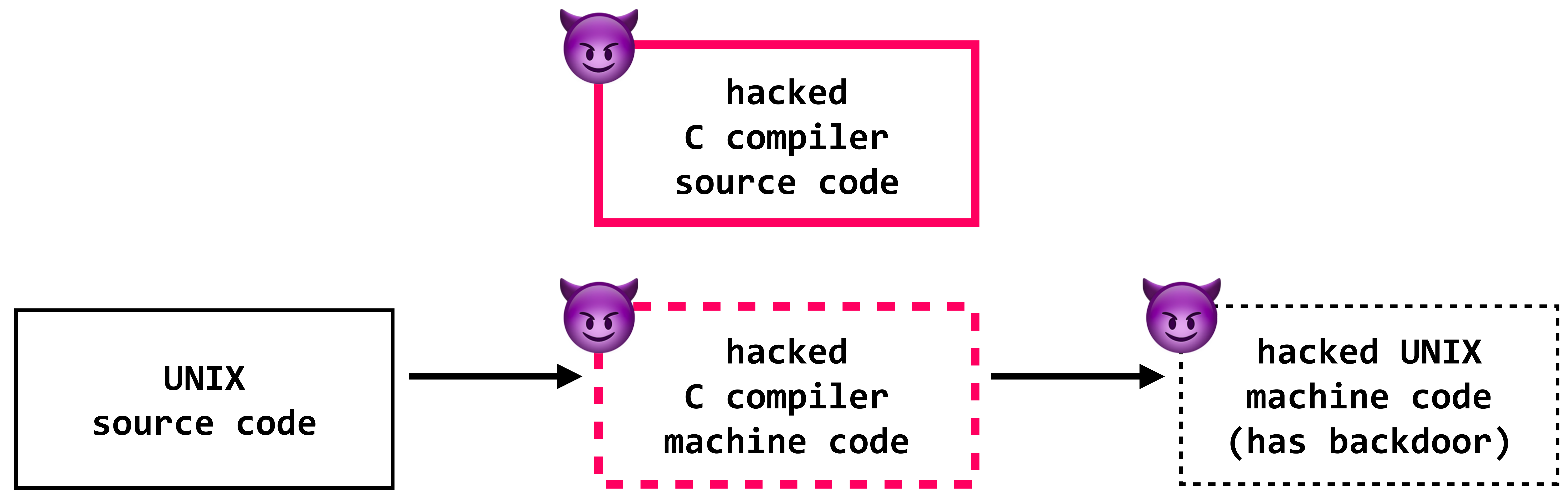
the hacked C compiler has code that
inserts a backdoor into UNIX

this backdoor *does not* exist in the UNIX source…

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code



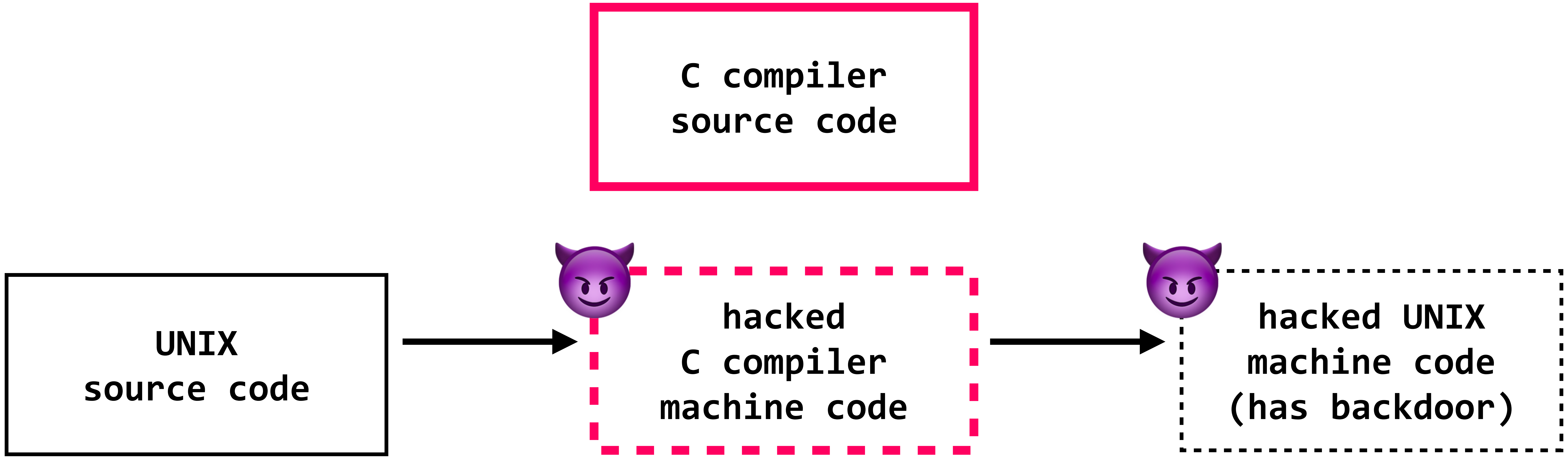
the hacked C compiler has code that
inserts a backdoor into UNIX

this backdoor *does not* exist in the UNIX source…
but it does exist in the hacked C compiler source

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code

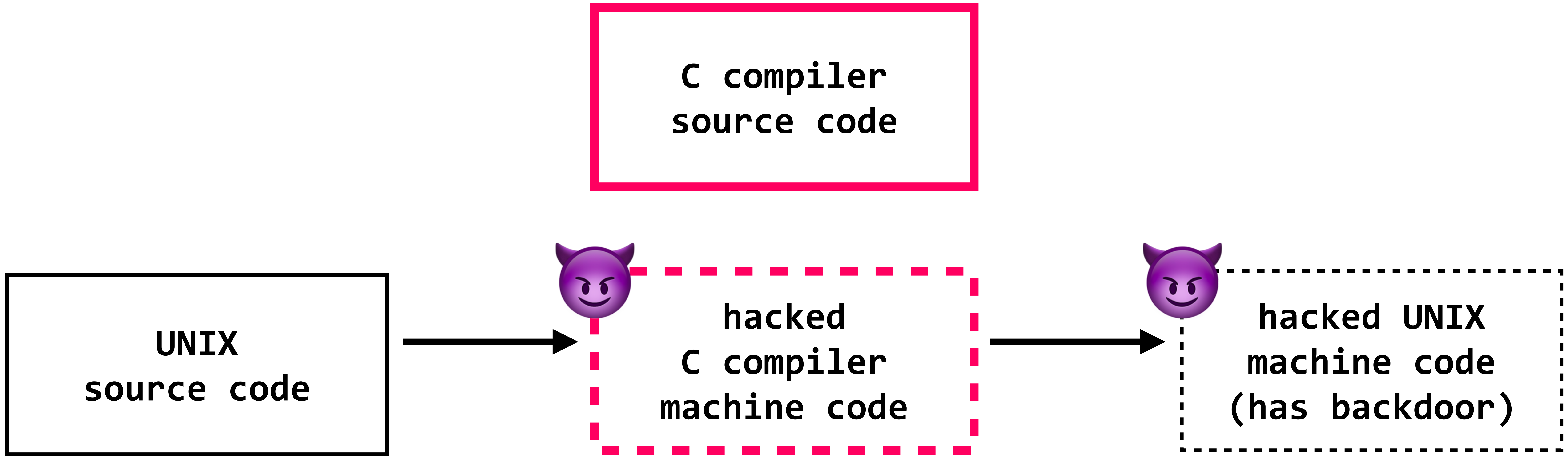


the hacked C compiler has code that *inserts* a backdoor into UNIX

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code



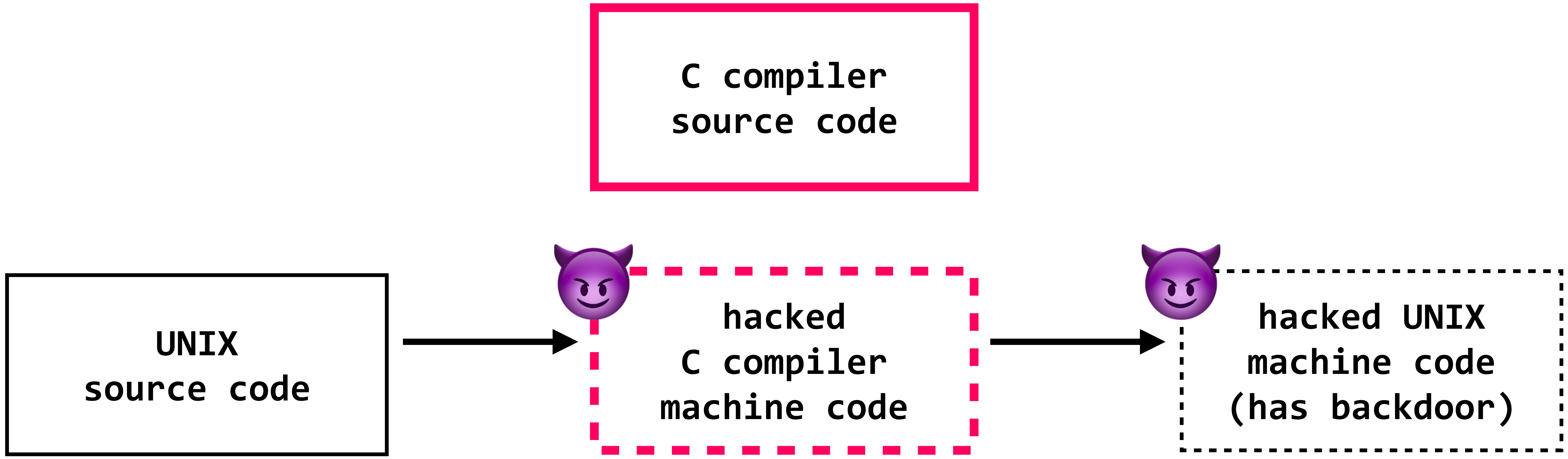
the hacked C compiler has code that
inserts a backdoor into UNIX

suppose the adversary lies, and tells you that the clean C compiler source is what generated the hacked C compiler; **can you detect this lie?**

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code



the hacked C compiler has code that *inserts* a backdoor into UNIX

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code

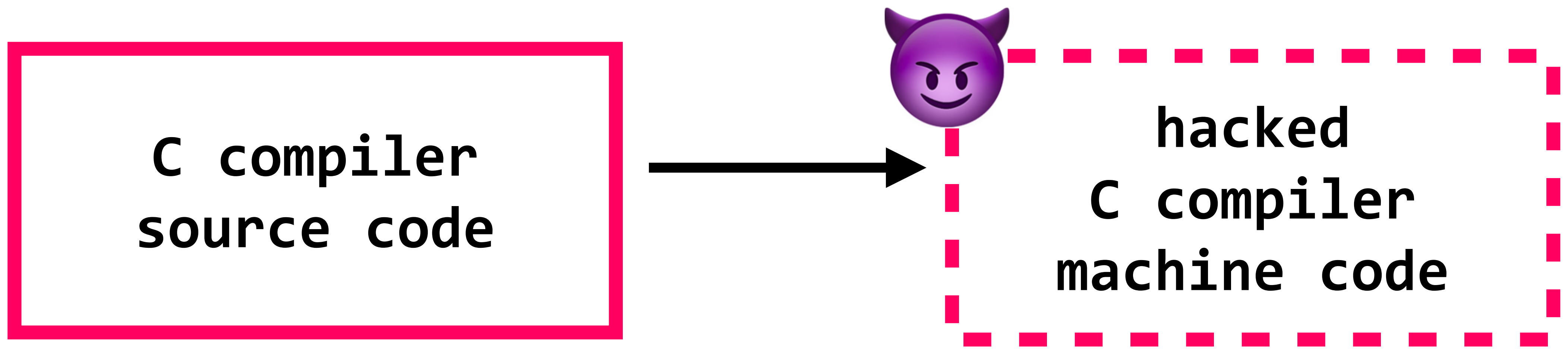
the hacked C compiler has code that
inserts a backdoor into UNIX

key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

compilers take source code as an input, and output machine code

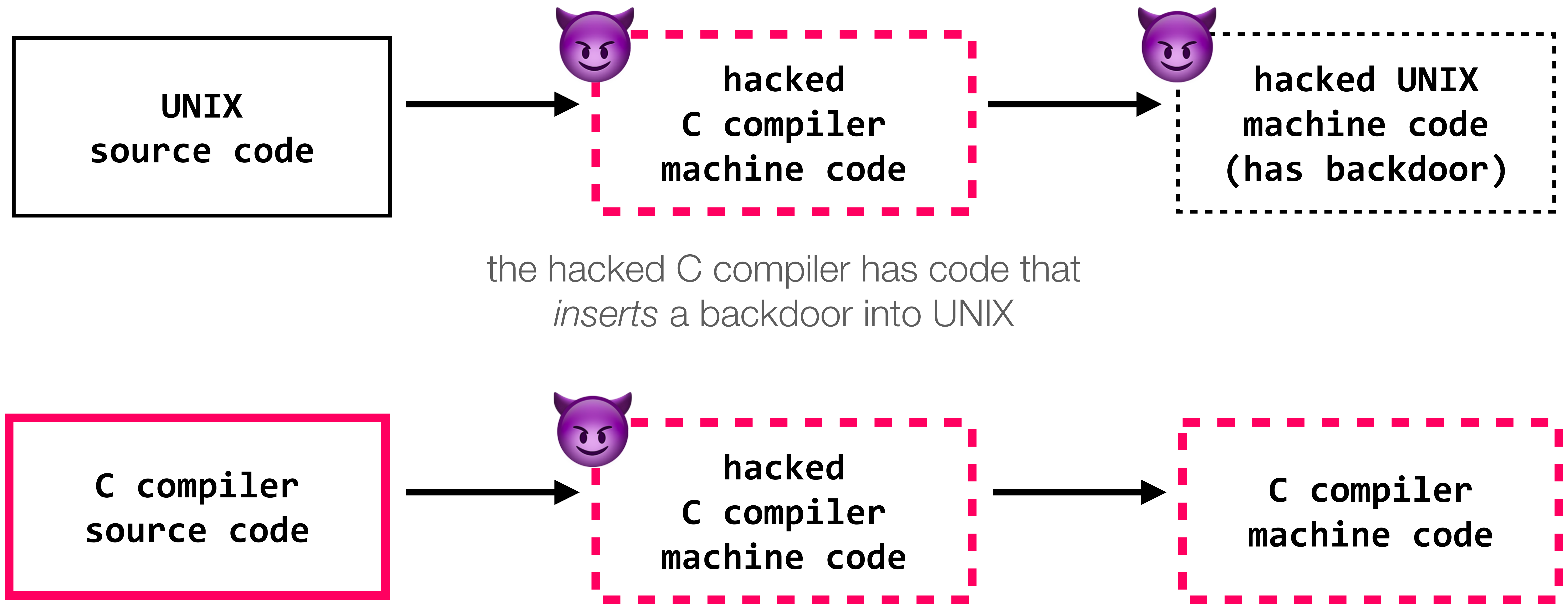
the hacked C compiler has code that *inserts* a backdoor into UNIX



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

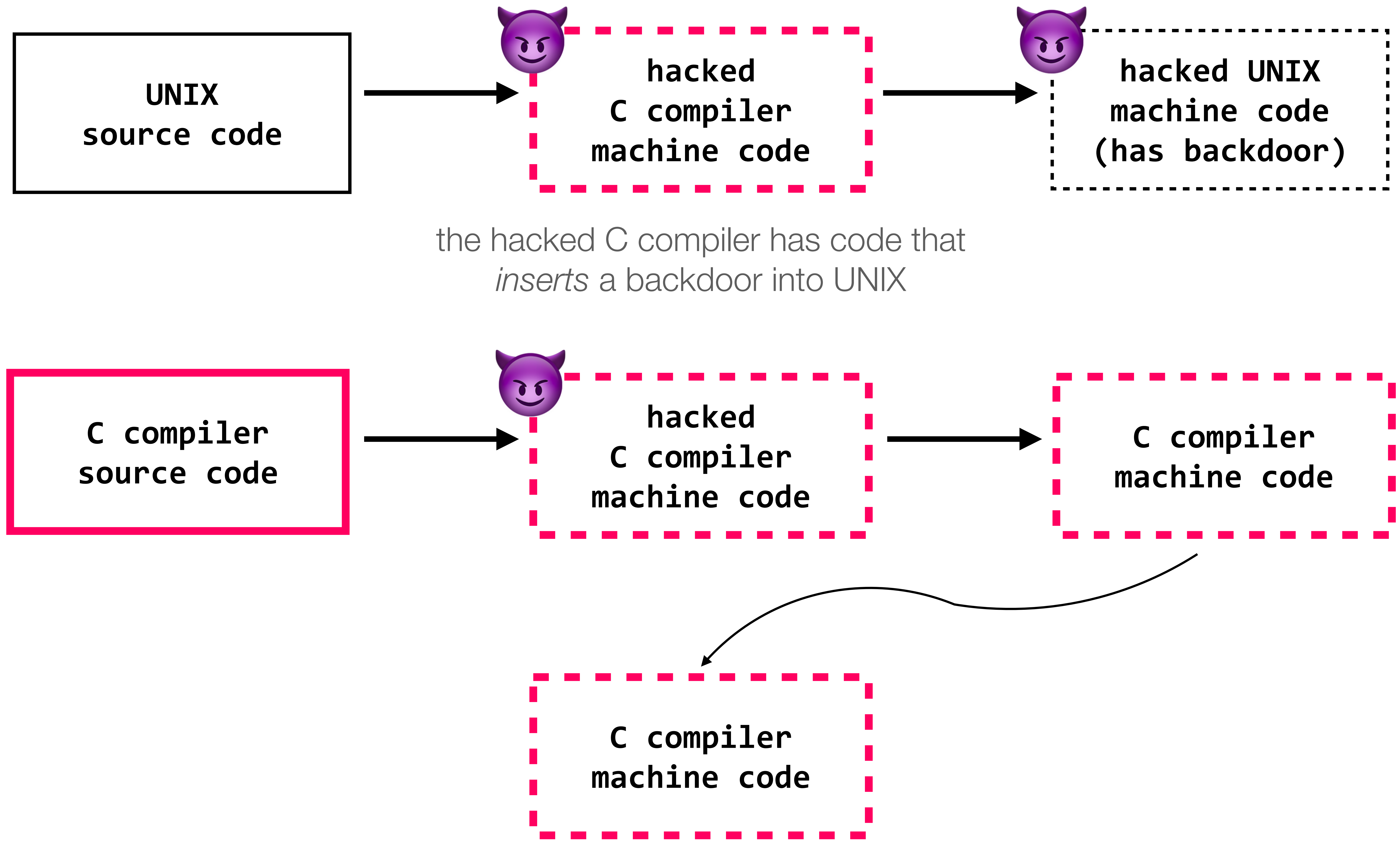
compilers take source code as an input, and output machine code



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

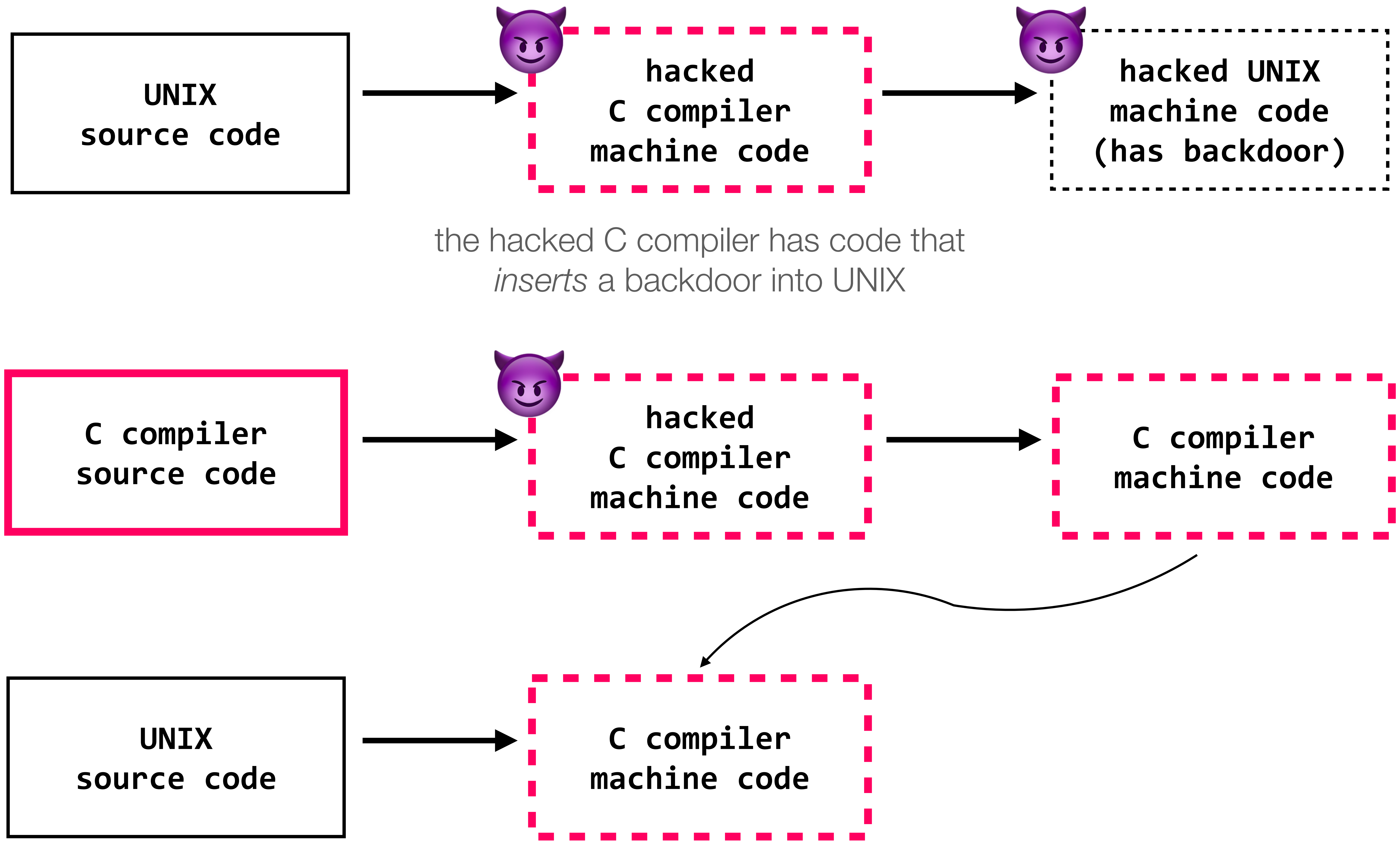
compilers take source code as an input, and output machine code



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

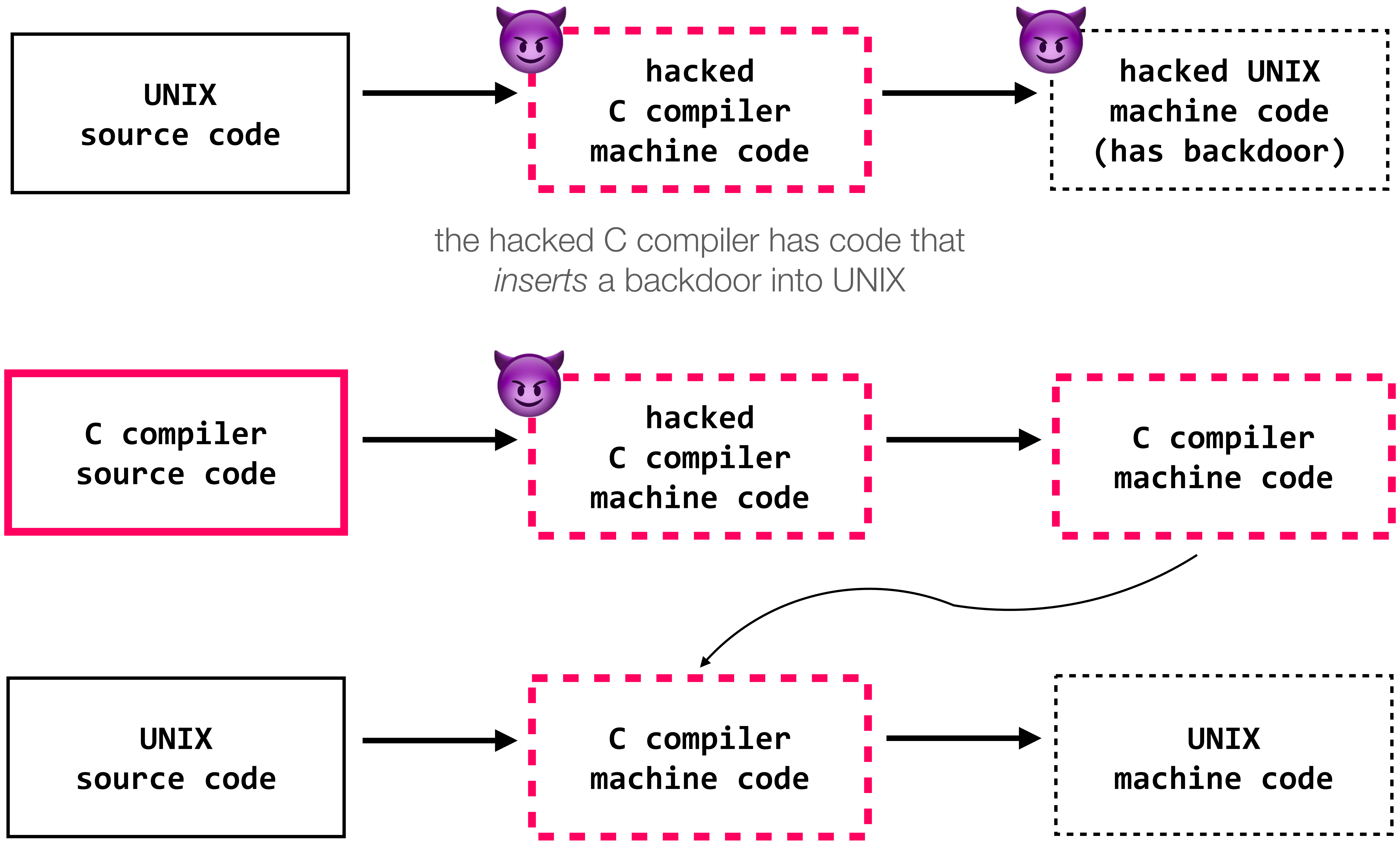
compilers take source code as an input, and output machine code



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

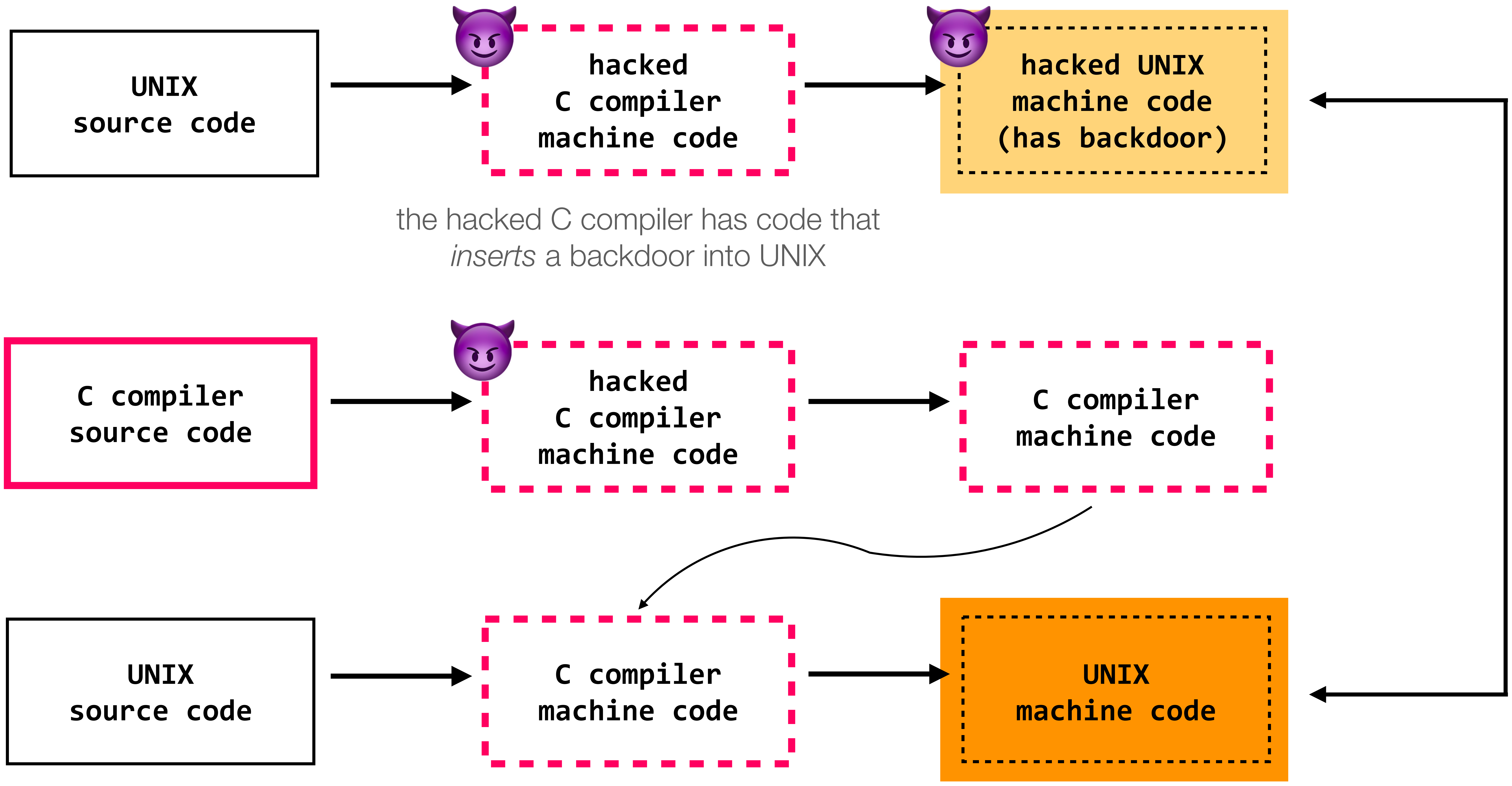
compilers take source code as an input, and output machine code



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

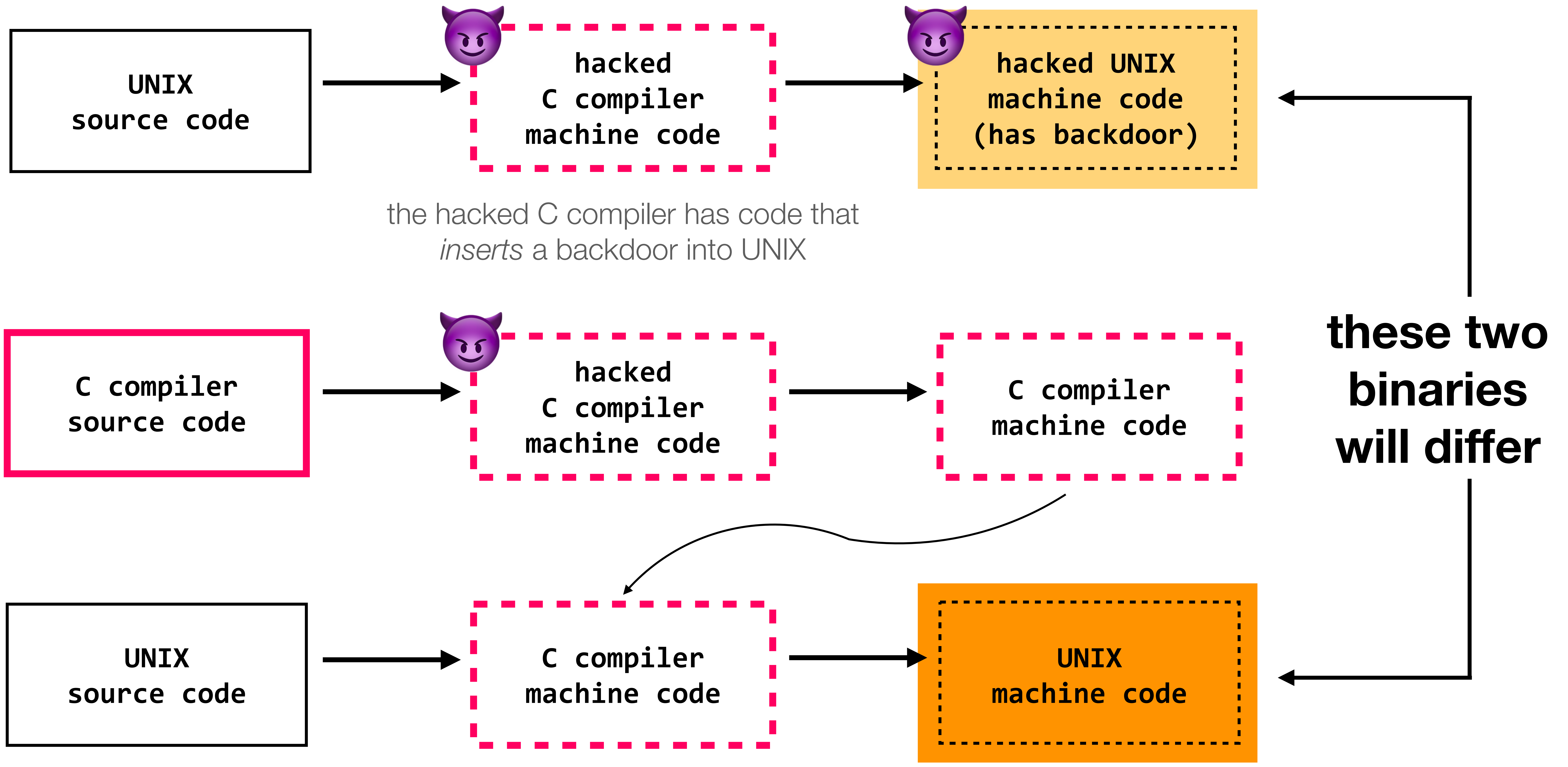
compilers take source code as an input, and output machine code



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

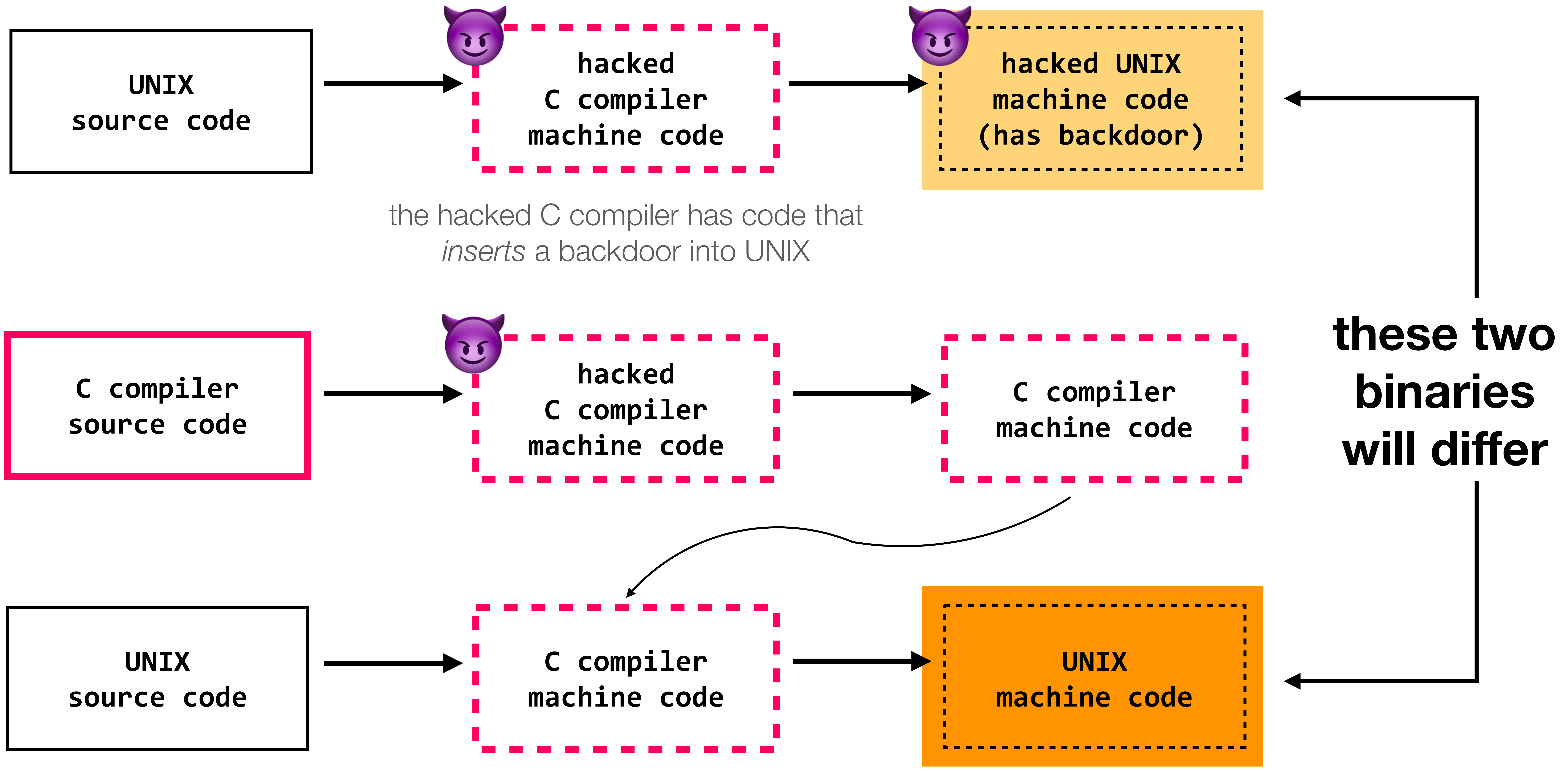
compilers take source code as an input, and output machine code



key point: we can determine whether source code is hacked by just reading code itself
(the code that inserts a backdoor would be obvious to someone familiar with the UNIX source)

compilers: can we trust them?

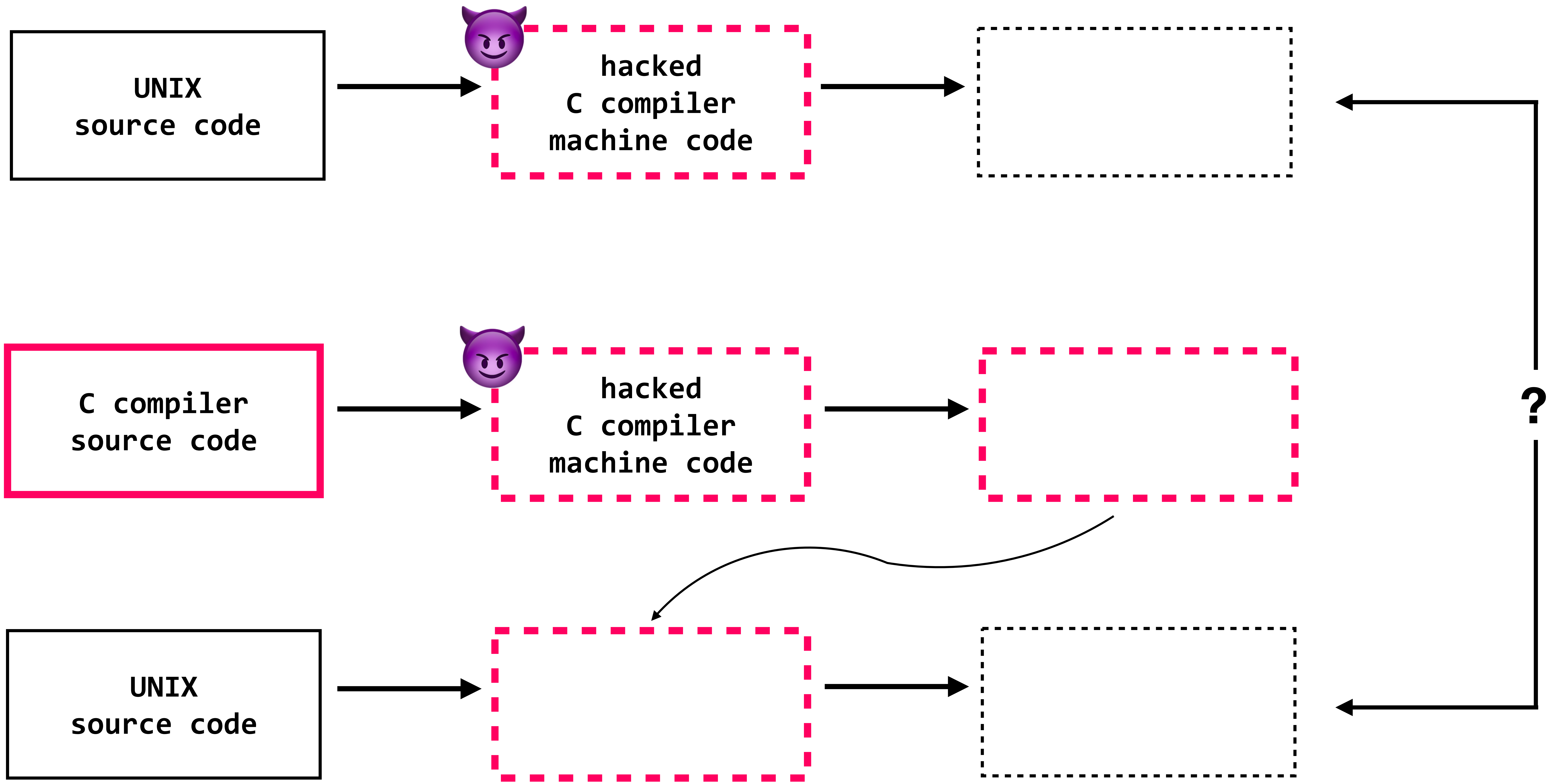
compilers take source code as an input, and output machine code



key point: we can detect a hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

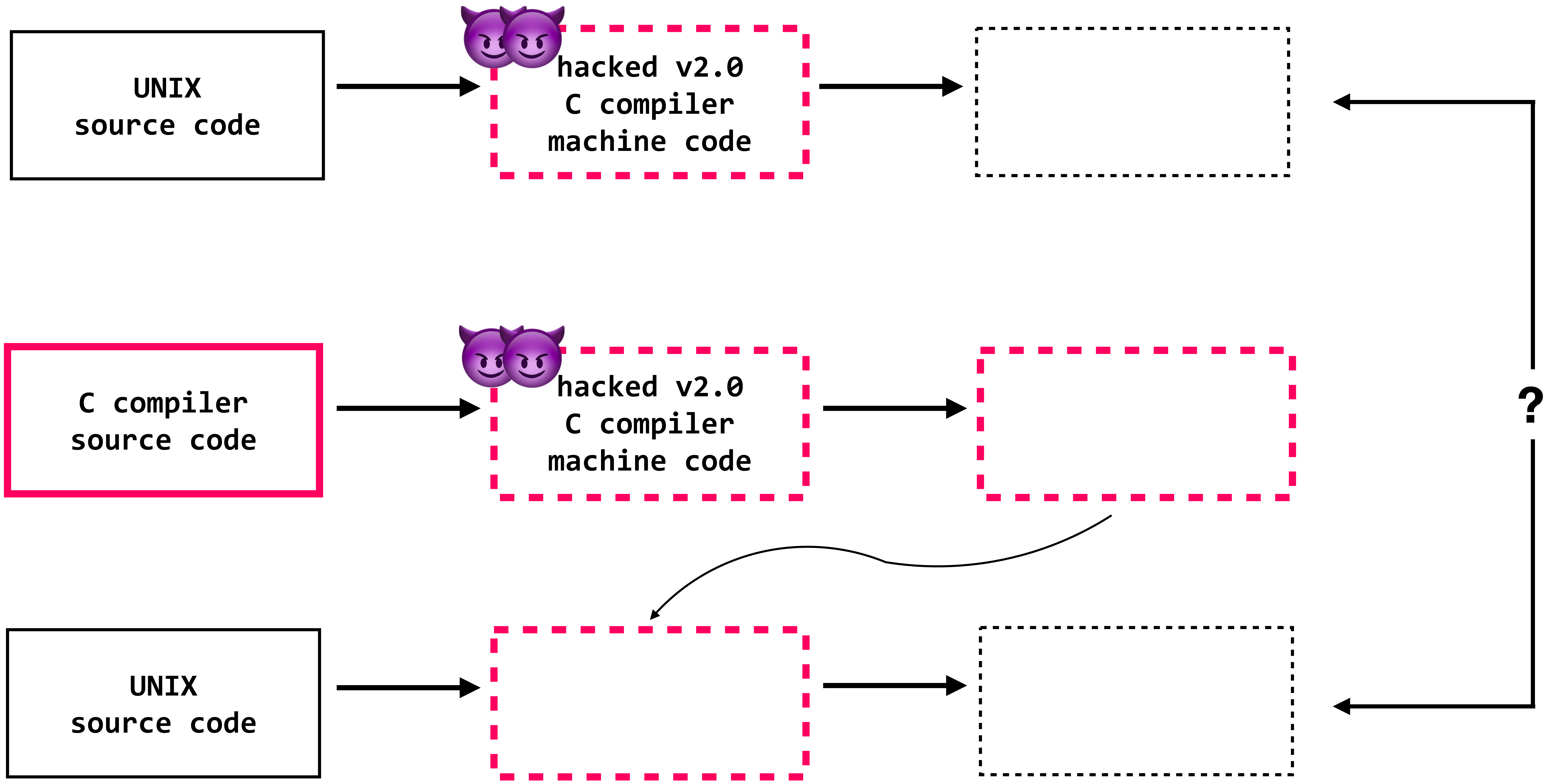


key point: we can detect a hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers

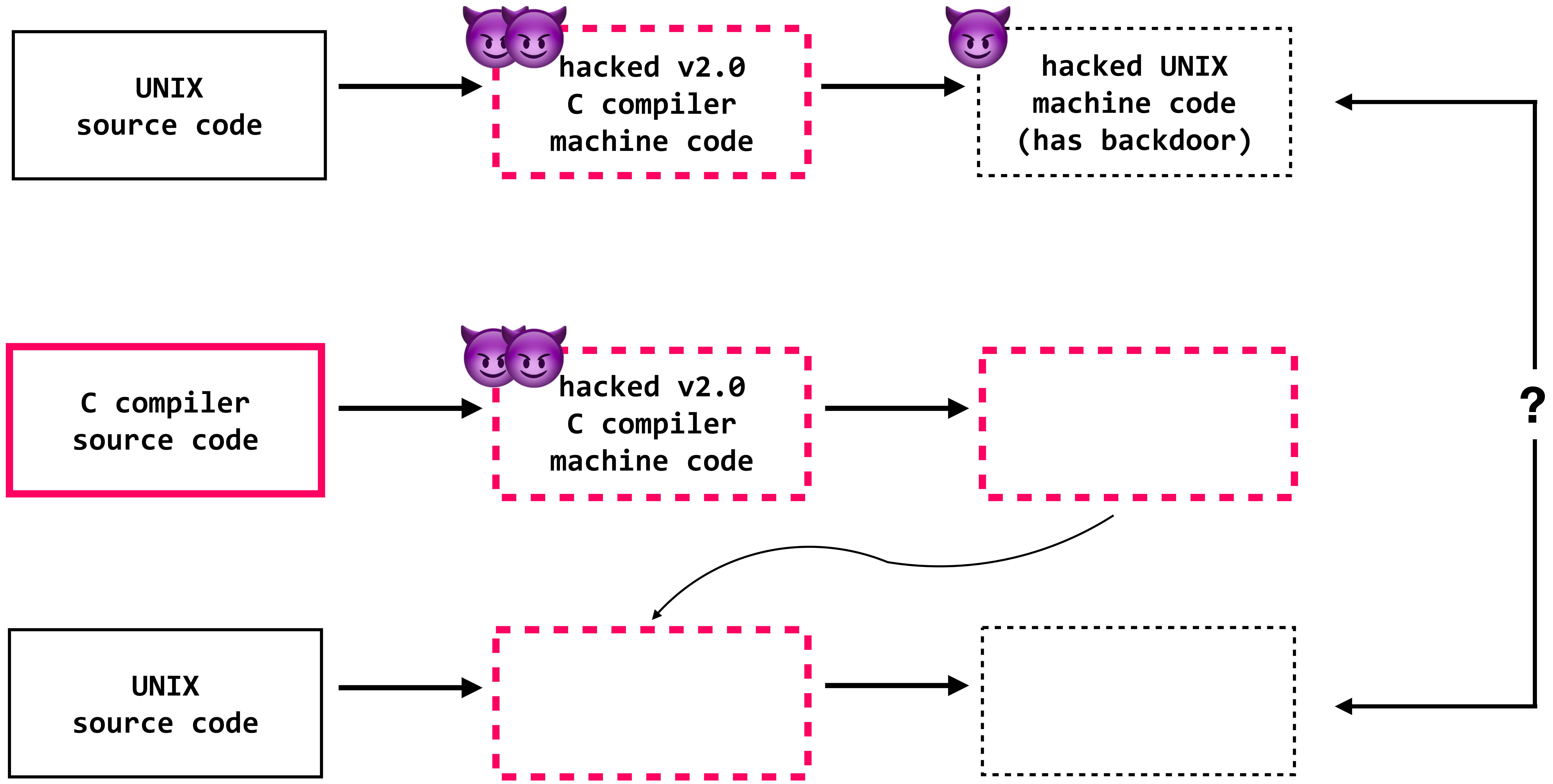


key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers

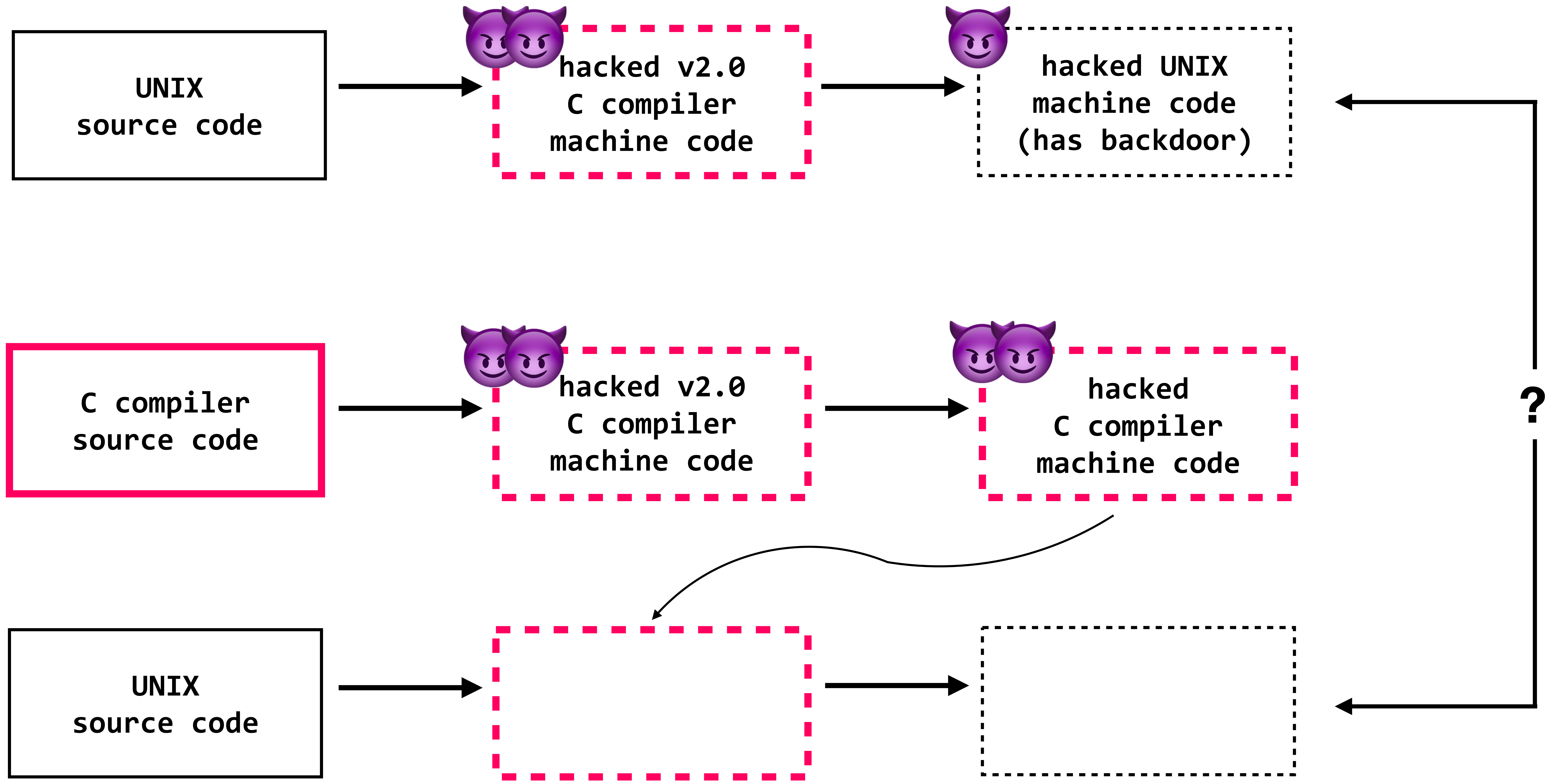


key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers

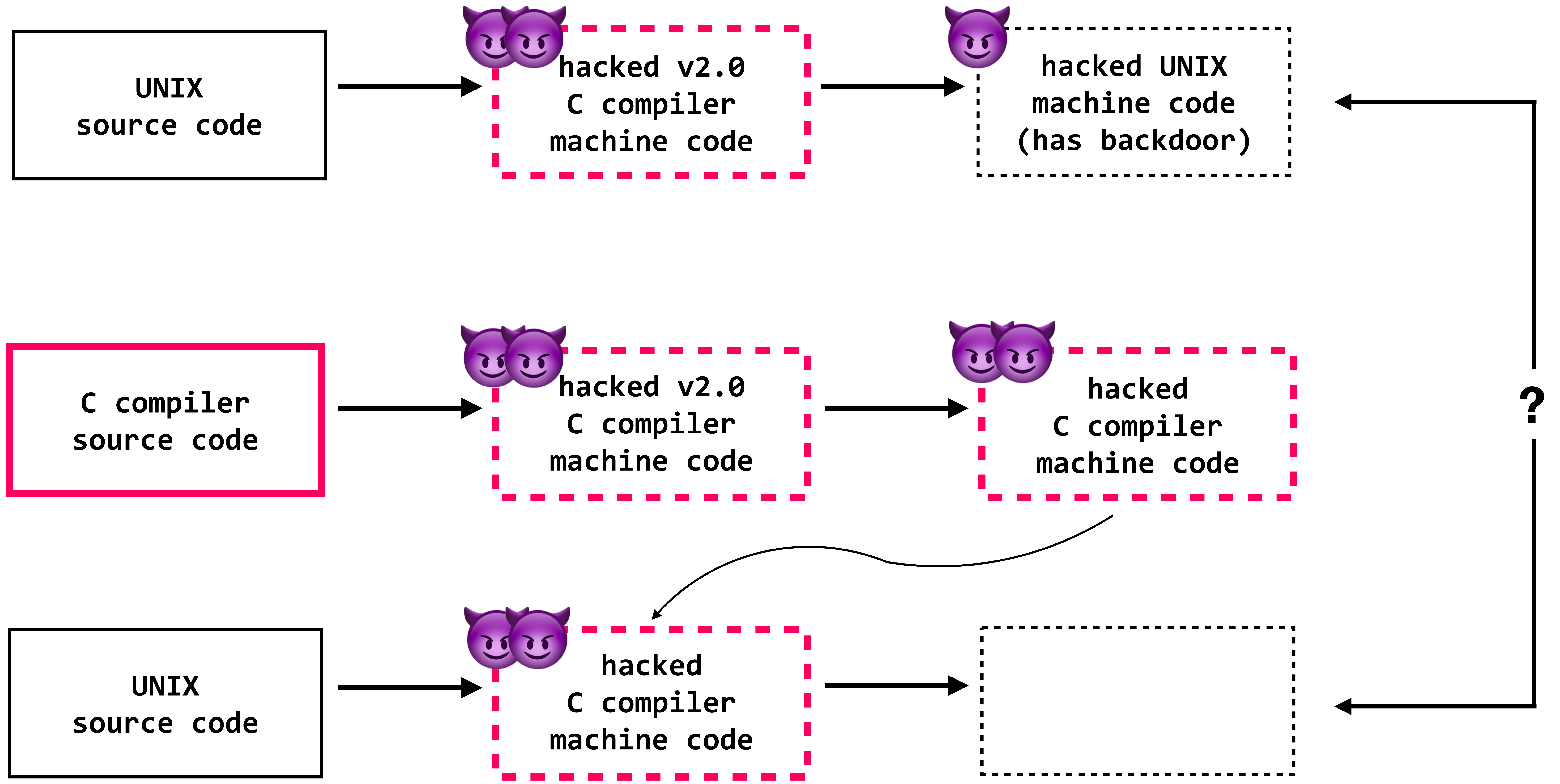


key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers

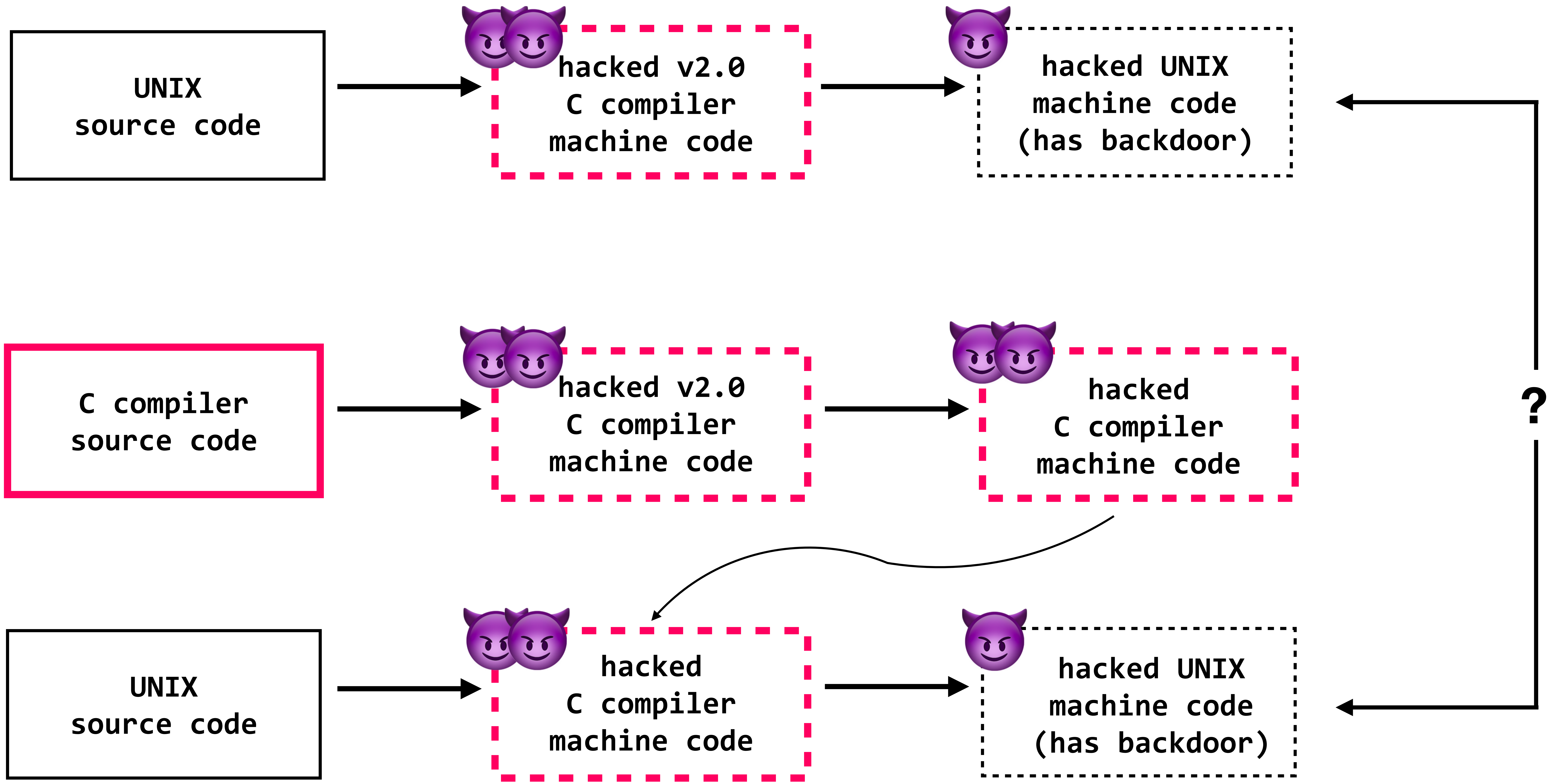


key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers

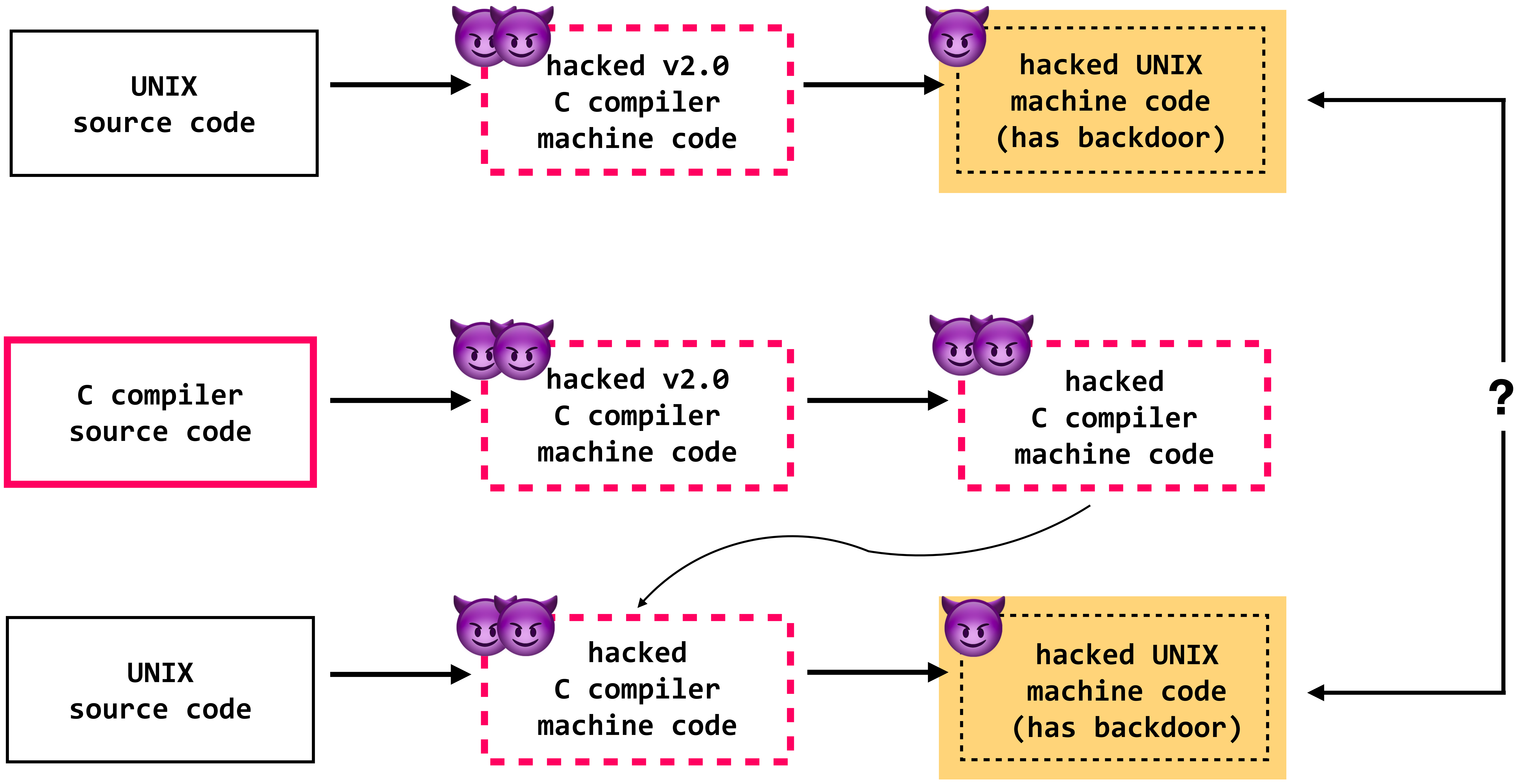


key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers

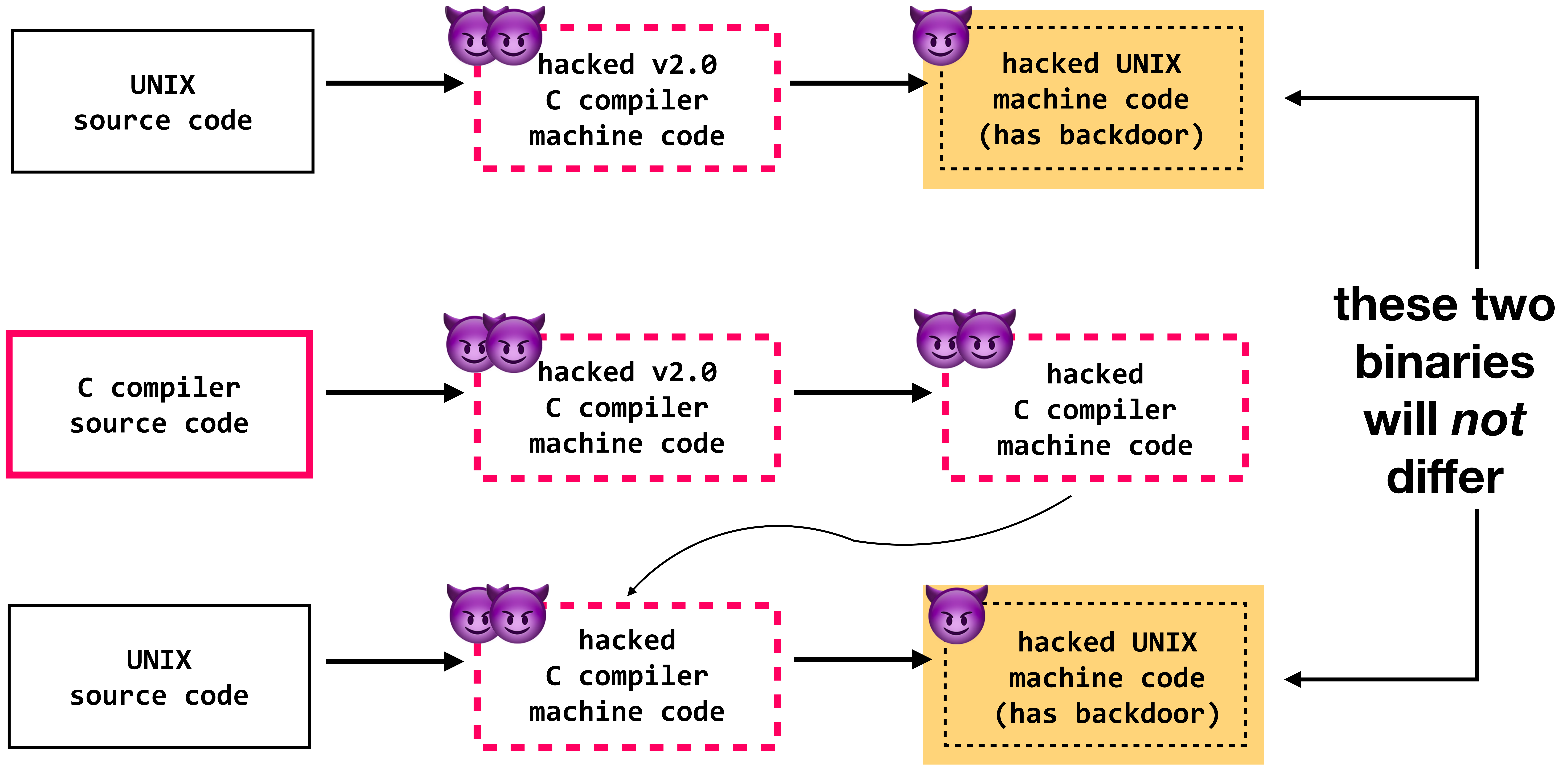


key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

the hacked v2.0 C compiler has code that *inserts* a backdoor into UNIX *and* code to insert backdoor-inserting code into C compilers



key point: we can detect ***the original*** hacked compiler by recompiling a clean compiler, using that to compile UNIX, and testing the output against what the hacked compiler produced

compilers: can we trust them?

compilers take source code as an input, and output machine code

REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM 17*, (July 1974), 365–375.
4. Unknown Air Force Document.

compilers: can we trust them?

compilers take source code as an input, and output machine code

REFERENCES

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-10. *Commun. ACM 15*, 3 (Mar. 1972), 135–143.
2. Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N.J., 1978.
3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. *Commun. ACM 17*, (July 1974), 365–375.
4. Unknown Air Force Document.

Karger, P.A., and Schell, R.R.
Multics Security Evaluation: Vulnerability Analysis
ESD-TR-74-193, Vol II, June 1974, page 52

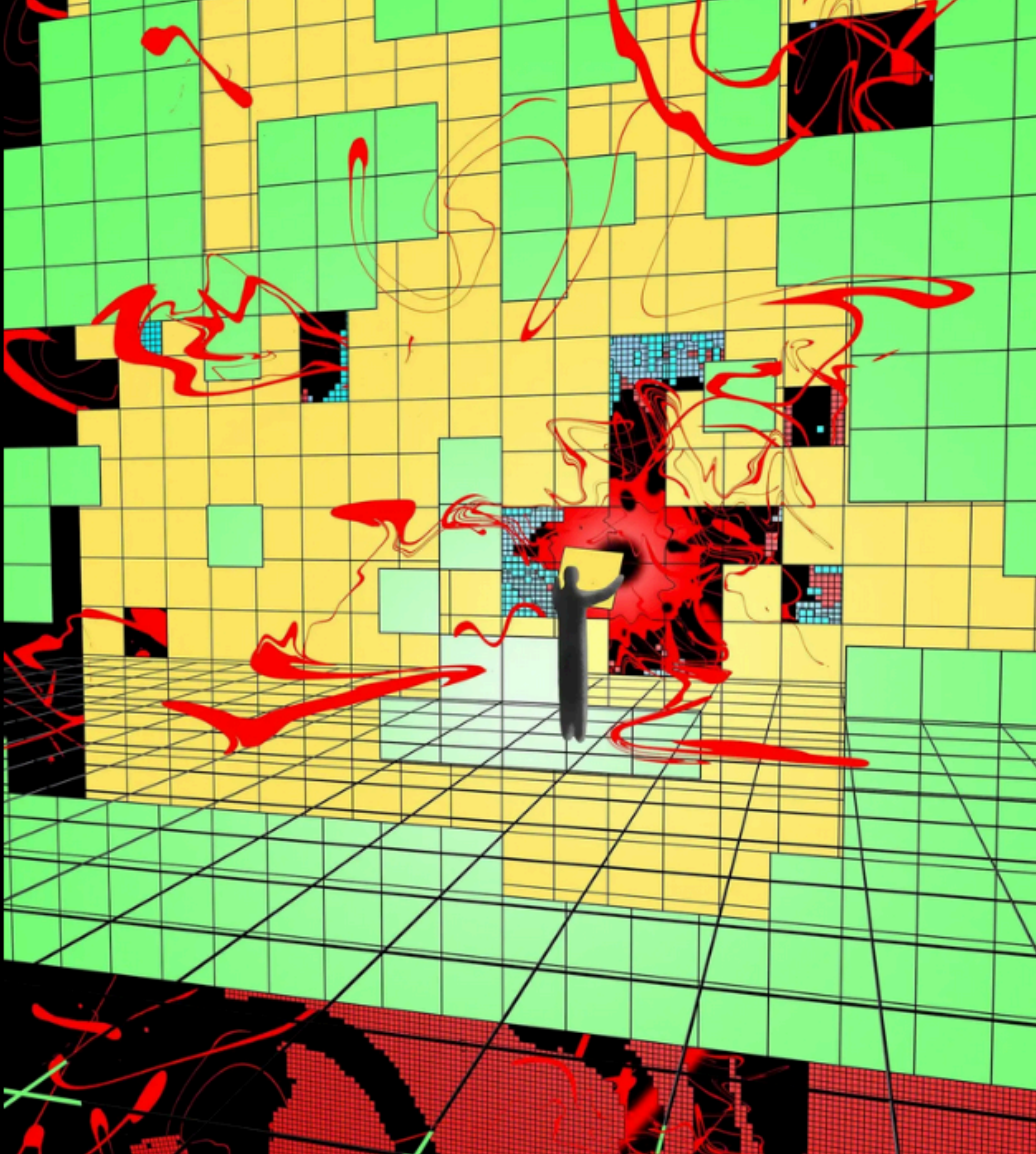
6.1800 in the news

THE SHIFT

Did One Guy Just Stop a Huge Cyberattack?

A Microsoft engineer noticed something was off on a piece of software he worked on. He soon discovered someone was probably trying to gain access to computers all over the world.

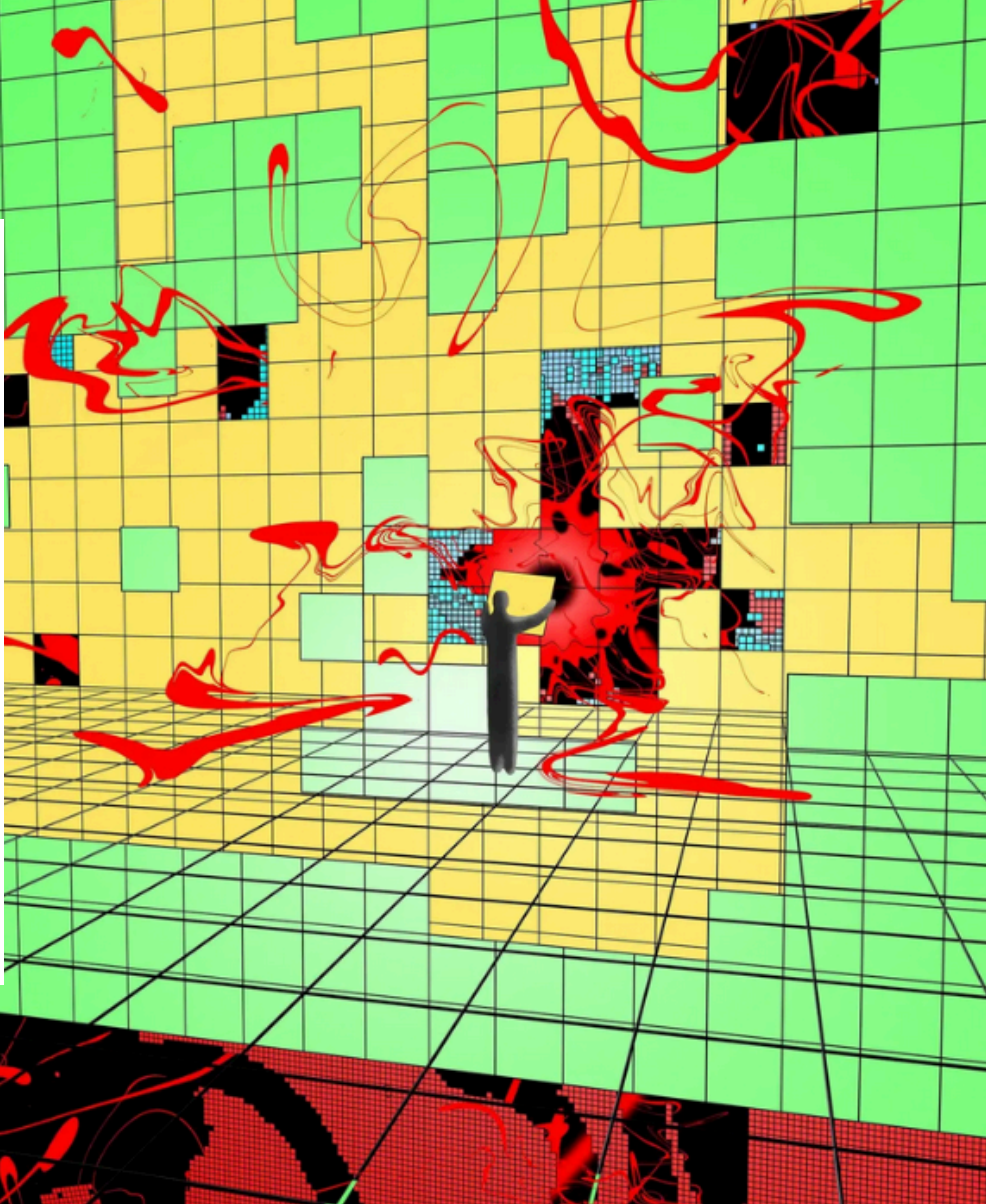
https://www.nytimes.com/2024/04/03/technology/prevent-cyberattack-linux.html



6.1800 in the news

According to some researchers who have gone back and looked at the evidence, the attacker appears to have used a pseudonym, “Jia Tan,” to suggest changes to xz Utils as far back as 2022. (Many open-source software projects are governed via hierarchy; developers suggest changes to a program’s code, then more experienced developers known as “maintainers” have to review and approve the changes.)

The attacker, using the Jia Tan name, appears to have spent several years slowly gaining the trust of other xz Utils developers and getting more control over the project, eventually becoming a maintainer, and finally inserting the code with the hidden backdoor earlier this year. (The new, compromised version of the code had been released, but was not yet in widespread use.)



https://www.nytimes.com/2024/04/03/technology/prevent-cyberattack-linux.html

low-level attacks can be insidious; as we implement solutions, there are often counter-attacks, and many solutions come at the cost of performance

however, just because we can't achieve perfect security does not mean that we cannot make progress; more sophisticated attacks are often more difficult for adversaries to carry out, and in some cases might not be worth the effort

while **thompson's "hack"** (attack?) illustrates to us that, to some extent, we cannot trust code we didn't write ourselves, it also advocates for **policy-based solutions** rather than technology-based

today's lecture + tomorrow's recitation should not stop you from ever touching a computer again